UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de escenarios para simulador multi-cámara basado en UNITY

Francisco Lobo García
Tutor: Dr. Juan Carlos San Miguel Avedillo
Ponente: Dr. José M. Martínez Sánchez

Junio 2018

Desarrollo de escenarios para simulador multi-cámara basado en UNITY

Francisco Lobo García
Tutor: Dr. Juan Carlos San Miguel Avedillo
Ponente: Dr. José M. Martínez Sánchez

Video Processing and Understanding Lab
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2018

Trabajo parcialmente financiado por el Ministerio de Economía y Competitividad del Gobierno de España bajo el proyecto TEC2014-53176-R (HAVideo) (2015-2017)

Resumen

El objetivo de este Trabajo de Fin de Grado (TFG) consiste en proporcionar un entorno virtual de pruebas con datos sintéticos, útil en la investigación de algoritmos en el ámbito de visión artificial (*Computer Vision*) donde se puedan simular diversas situaciones en una gran variedad de entornos y condiciones, algo difícilmente replicable en el mundo real. Por lo que este TFG presenta el desarrollo que se debe llevar a cabo para la realización de escenarios virtuales basados en el motor gráfico de Unity 3D, los cuales puedan ser integrados en un sistema distribuido multicámara basado en este motor gráfico, en concreto, en el simulador diseñado e implementado por Mario González en 2017 para el laboratorio VPU de la Universidad Autónoma de Madrid llamado: *Multi-camera System Simulator* (MSS). A lo largo del presente documento se detalla el diseño y la implementación seguida para desarrollar un escenario que simule cruces de calles de una ciudad donde ocurren numerosos eventos y situaciones de interés tales como coordinación, colisiones y acciones indebidas entre personas y vehículos. Combinando el simulador MSS con el escenario propuesto se proporciona una alternativa en la investigación en *Computer Vision*, con el fin de solventar las limitaciones técnicas y flexibles de los entornos de pruebas más comunes. Por último, se realiza una evaluación del comportamiento de los recursos computacionales comparando este nuevo escenario frente al escenario base disponible en el simulador MSS, para seguidamente discutir los resultados y descubrir los límites técnicos del escenario.

Palabras clave

Visión Artificial, simulación, escenario, evento de interés, situación de interés, modelo 3D, Unity, tienda de recursos de Unity, Sistema Simulador Multi-cámara, MSS, Escuela Politécnica Superior, EPS, cruce de calles, ciudad, frames por segundo, FPS.

Abstract

The objective of this Final Degree Thesis is to provide a virtual testing environment with synthetic data, useful in the investigation of algorithms for Computer Vision, where different situations can be simulated in a wide variety of environments and conditions, something difficult to replicate in the real world. Therefore, this proyect presents the creation of virtual scenarios based on the Unity 3D graphics engine and the integration in a distributed multi-camera system based on this graphic engine, specifically, in the simulator designed and implemented by Mario González in 2017 for the VPU laboratory of the Autonomous University of Madrid called: Multi-camera System Simulator (MSS). Throughout this document, we describe the design and implementation to develop a scenario simulating crossroads in a city environment where numerous situations of interest happen, such as coordination, collisions and illegal actions between people and vehicles. Combining the MSS simulator with the proposed scenario provides an alternative in the research of Computer Vision, in order to solve the technical and flexible limitations of the most common testing environments. Finally, an evaluation of the computational resources required by the city scenario is performed to understand its technical limits whilst comparing this new scenario with the base one available in the MSS simulator.

Keywords

Computer Vision, simulation, scenario, interest's events, interest's situations, 3D model, Unity, Unity Asset Store, Multi-camera System Simulator, MSS, Higher Polytechnic School, EPS, crossroads, city, framerate, FPS.

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor del TFG, Juan Carlos San Miguel Avedillo, su ayuda, tiempo, disponibilidad, motivación, conocimiento y profesionalidad a lo largo de este año de duro trabajo, mostrándome y aportándome su entusiasmo y ganas de seguir aprendiendo, generando en mi, gran interés por esta rama de la informática, desconocida para mi a lo largo de la carrera. De esta forma, me ha dado a conocer este ámbito y aplicaciones de la informática, además de trabajar junto con el grupo de investigación del laboratorio VPU, departamento que se encuentra en el edificio C de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid.

También me gustaría agradecer y dedicar a mis padres, Paco y Herme, este trabajo, por el gran apoyo económico y emocional que he necesitado todos estos años mientras cursaba el grado y que de otra forma hubiera sido imposible de llevar a cabo. Agradecer a mis hermanos Borja y Carol por transmitirme ese afán que tienen por superarse a sí mismos día tras día. A mis abuelos, Tita y Fernando, que sin entender actualmente que hago y a que me dedico realmente después de haber estado toda una vida estudiando, siguen ahí, apoyándome y dándome todo su cariño.

Es necesario agradecer a los compañeros del grado como Guille, Iván, René y Kiko, entre otros, la gran paciencia que han tenido en clase conmigo, y los buenos ratos a lo largo de estos últimos años que hemos pasado juntos, y que han hecho más amenos los agobios y complicaciones de asignaturas y prácticas. Agradecer el apoyo incondicional que me ha dado mi fiel compañera, Nadia, por escuchar, sin quedarse dormida o aburrirse por no entender ni una sola palabra, cuando la hablaba de asignaturas o trabajos de mi grado.

Y por último, y no menos importante, me gustaría agradecer a mis amigas y exprofesoras del instituto, Paloma, Pilar y Teresa, las cuales no se dedican a impartir asignaturas, si no que transmiten mucho más: pasión por su profesión, enseñanza, trato humano y conocimiento.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

La visión artificial (o más conocida como *Computer Vision*) es un área interdisciplinaria con el objetivo de comprender el mundo real, a partir de imágenes o vídeos, mediante algoritmos informáticos. Para ello es necesario, que estos algoritmos adquieran, procesen, analicen y comprendan dichas imágenes y fotogramas, y a partir de ellas, extraigan datos e información, útil para ser tratada por un ordenador, simulando la forma en como los humanos captamos dichas imágenes con los ojos y empleamos el cerebro para procesar dicha información del mundo que nos rodea. En los últimos años, ha crecido en gran medida el uso de *Computer Vision* en numerosas áreas científicas [1], convirtiéndose en una herramienta de nuestra vida cotidiana, debido al enorme potencial que tiene para cambiar nuestro estilo de vida, estando presente en diversas áreas, que van desde la vídeovigilancia, multimedia y visión artificial, a las áreas emergentes como asistencia al conductor, cirugía guiada por visión, robótica asistida y en pesca marina [2]. En la Figura 1.1, se muestra ejemplos de diversas aplicaciones de *Computer Vision* en la vida cotidiana (e.g monitorización de tráfico, videoseguridad, asistencia en carretera).

Figura 1.1: Ejemplo de aplicaciones en distintas áreas de *Computer Vision*. (https://goo.gl/zztU2i) (https://goo.gl/iqSx7z) [3]

La complejidad de *Computer Vision* es debida a diversos factores tales como la gran diversidad de contenido visual, requisitos de tiempo real, recursos limitados y comunicación entre sistemas distribuidos. Por ello, el uso de simuladores y entornos controlados, ha sido de utilidad para favorecer el desarrollo de algoritmos y aplicaciones antes de un despliegue en aplicaciones reales. Lo cual ayuda considerablemente en la evaluación y captación de datos e información del mundo real donde, puede necesitarse reproducir con diferentes parámetros

(e.g. densidad de personas y vehículos, complejidad de las acciones, posicionamiento, movimiento o cantidad de cámaras) las situaciones de interés tantas veces como sea necesario, con el fin de evaluar y entrenar los algoritmos. Los datos del mundo real no permiten repetición y modificación de condiciones para el testeo de algoritmos. Por ese motivo las herramientas de simulación proporcionan entornos de prueba flexibles donde emular escenarios con máximo detalle. Los simuladores permiten configuración dinámica del sistema adecuándose a un algoritmo particular o a propiedades requeridas en cada momento.

Los simuladores deben imitar con gran precisión y detalle las condiciones reales para que el rendimiento del algoritmo pueda ser extrapolado a sistemas del mundo real, además de permitir determinar la eficacia y eficiencia de un sistema antes de ser desplegado, reduciendo considerablemente los costes y los tiempos de instalación, necesitando únicamente un equipo donde instalar y ejecuta el simulador.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado consiste en diseñar y desarrollar escenarios en los que se ocurran eventos y situaciones de interés, los cuales tengan la capacidad de ser importados al sistema multicámara distribuido (MSS) del VPUlab. El sistema virtual MSS es capaz de administrar varias cámaras distribuidas en tiempo real, configurar las propiedades de la cámara y dar la posibilidad de transmitir datos. El motor gráfico Unity 3D[1] es nuestro punto de partida.

Para alcanzar el objetivo principal, se definen los siguientes subobjetivos:

- Estudiar escenarios multicámara de interés, en los cuales se produzcan numerosos eventos que sean relevantes.
- Estudiar el estado del arte, incluyendo los simuladores multicámara relacionados y motores gráficos disponibles para motivar la elección Unity 3D.
- Realizar una búsqueda, identificación y elección de recursos disponibles, así como, modelos 3D y algoritmos de inteligencia artificial de utilidad.
- Definir y formalizar la generación de elementos estáticos de los escenarios, los cuales conforman la extensión y la ambientación de dicho escenario.
- Definir y formalizar la generación de elementos dinámicos de los escenarios, así como, los eventos e interactividades que realizarán dentro de un escenario.
- Generación e implementación de escenarios con elementos estáticos y dinámicos a distintas calidades y complejidades.
- Integrar los escenarios desarrollados en el simulador MSS, para finalmente, evaluar su funcionalidad y su rendimiento.

[1] https://unity3d.com/es

1.3. Organización de la memoria

La memoria consta de los siguientes capítulos:

- Capítulo 1. Este capítulo presenta la motivación y los objetivos de este Trabajo de Fin de Grado.
- Capítulo 2. Este capítulo aporta una descripción general de los trabajos relacionados y una comparativa entre los distintos motores gráficos.
- Capítulo 3. Este capítulo describe el diseño seguido para introducir distintos eventos y situaciones de interés en el escenario virtual.
- Capítulo 4. Este capítulo describe la implementación de las distintas funcionalidades desarrolladas.
- Capítulo 5. Este capítulo presenta y analiza los resultados de diversas pruebas.
- Capítulo 6. Este capítulo resume los principales logros de este trabajo y aporta sugerencias para futuros trabajos.
- Bibliografía y Glosario.

Para proporcionar explicaciones con mayor detalle, se incluye un apéndice:

- Apéndice A. Este apéndice proporciona más información sobre los motores gráficos más relevantes del mercado.
- Apéndice B. Este apéndice proporciona más información sobre los simuladores virtuales más relevantes.
- Apéndice C. Este apéndice muestra el estudio completo de los recursos disponibles que pueden ser de utilidad para este proyecto.
- Apéndice D. Este apéndice proporciona tutoriales para la creación de objetos.

Capítulo 2

Estado del arte

Este capítulo analiza trabajos sobre simulación de sistemas con múltiples cámaras En la sección 2.1, se describen los distintos motores gráficos más relevantes. Seguidamente, en la sección 2.2, se estudian simuladores relacionados con este proyecto.

2.1. Motores gráficos

En los últimos años, la industria de los videojuegos y de la realidad aumentada ha crecido en tal medida, que las simulaciones interactivas y las representaciones gráficas detalladas no necesitan ejecutarse en costosos y especializados equipos. Hoy en día, cualquiera podría ejecutarlas en su propio ordenador, videoconsola, o incluso dispositivo móvil, a un costo significativamente menor. Esto es posible debido al uso de arquitecturas estandarizadas (*frameworks*) donde los juegos y simuladores virtuales pueden ser diseñados con mayor optimización y facilidad. Para investigación, es necesario estudiar que motor gráfico es el más apropiado para una tarea concreta, ya que no todos ofrecen las mismas características.

Los motores gráficos estudiados comparten especificaciones, pero tienen diversas características que es interesante comparar. Shiva 3D, Torque 3D, Unreal Engine y Unity 3D tienen una página oficial con complementos, modelos 3D, sistemas y herramientas, tanto gratis como de pago. Algunos de los motores gráficos estudiados han sido utilizados anteriormente en investigación, como SLNCR que utiliza Panda 3D, OVVV que trabaja sobre *Source Engine*, UnrealCV que extiende Unreal Engine y múltiples simuladores que funcionan sobre Unity 3D. En la Tabla 2.1 se muestra una comparativa de los simuladores. Para mayor información, se proporcionan descripciones adicionales en el Apéndice A.

2.2. Simuladores virtuales

En *Computer Vision*, los simuladores virtuales proporcionan un entorno de pruebas para el diseño y desarrollo de algoritmos. Las imágenes y vídeos sintéticos a menudo se han considerado inadecuados para medir el rendimiento de los algoritmos [4], ya que, eliminaban características y detalles que las cámaras digitales eran capaz de captar y no ofrecían una suficiente simulación realista. Pero en los últimos años, con el uso de estos *frameworks*, es

Tabla 2.1: Tabla comparativa de los motores gráficos más relevantes.

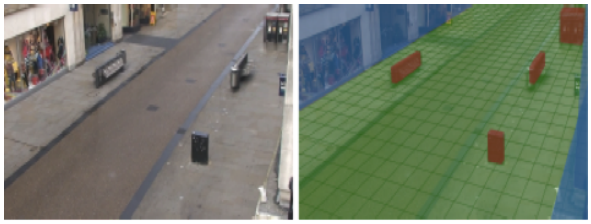
Motor Gráfico	**Licencia Gratis**	**Código libre**	**Página recursos**	**Otros Servicios**	**Lenguaje de Scripts**	**Uso investigación**
Panda 3D	Si	Si	No	Compatible VR Pirepheral Net.	C++ y Python	Si
Ogre 3D	Si	Si	No	Compatible Kinect	C++	No
Quest 3D	No	No	No	-	C++ y Lua	No
ShiVa 3D	No	No	Si	Plugin web	Lua	No
Torque 3D	No	Si	Si	-	C++	No
Source Engine	Si	Si	No	-	C y C++	Si
Cry Engine	Si	Si	Si	Compatible Kinect	Python, C#, C++ y Lua	No
Amazon Lumberyard	Si	Si	No	Servicio web de Amazon	C++	No
Unreal Engine	Si	Si	Si	-	C++	Si
Unity 3D	Si	No	Si	Ads, informe rendimiento	C#, Mono y JavaScript	Si

posible simular escenarios fotorrealistas con el fin de diseñar, desarrollar, probar y ahorrar costes y tiempos significativos. Existen diversos simuladores virtuales que cuentan con características muy diferentes, de tal forma, que pueden ser clasificados en función de los datos que utilizan para generar sus escenarios, es decir, si necesitan abastecerse de datos reales para generar los entornos simulados, o si generan y personalizan sus escenarios tal y como quieran sus desarrolladores, sin limitación alguna y sin datos reales. Para mayor información, se proporcionan descripciones adicionales en el Apéndice B.

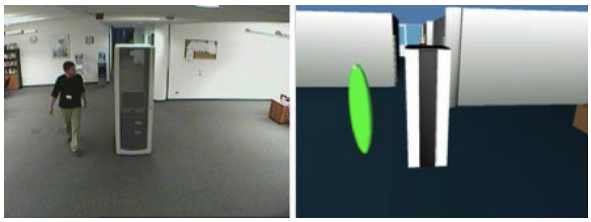
Los simuladores a partir de datos reales obtenidos de grabaciones del mundo real, aportan situaciones más auténticas puesto que recrean eventos fielmente a como ocurrieron en la realidad. El inconveniente es que se limita la personalización de los escenarios utilizados ya que se necesita la situación real. Ejemplos son Virtual Pedestrian (Figura 2.1a), 3DWM (Figura 2.1b), y Virtual KITTY (Figura 2.1c).

Los simuladores generados sintéticamente permiten a sus desarrolladores personalizar libremente los escenarios, creando y generando nuevas situaciones y pudiendo configurar libremente diversas condiciones del escenario. En estos simuladores, hay que realizar un gran esfuerzo para que las escenas y eventos se parezcan a la realidad, siendo de mayor utilidad en el entrenamiento de algoritmos. Ejemplos son COnGRATS (Figura 2.2a), SLNCR (Figura 2.2b), PHAV (Figura 2.2c), y OVVV (Figura 2.2d).

Los simuladores estudiados en el apéndice B son útiles para su uso en investigación, pero no todos tienen las mismas características, y por lo tanto, es necesario estudiar cual es el más apropiado para una tarea o finalidad específica. En primer lugar, Virtual Pedestrian, MSCV, UnrealCV, Virtual KITTI, SceneNET RGB-D y PHAV no permiten colocar múltiples cámaras, además de que tanto estos simuladores, como COnGRATS no están preparados



(a) Virtual Pedestrian. [5]



(b) 3DWM. https://goo.gl/4zgNYH

(c) Virtual KITTI. [6]

Figura 2.1: Ejemplos de simuladores generados a partir de escenarios reales.

(a) COnGRATS. https://goo.gl/eh4Vpc

(b) SLNCR. [7]

(c) PHAV. [8]

(d) OVVV. [9]

Figura 2.2: Ejemplos de simuladores generados a partir de escenarios sintéticos.

para trabajar en tiempo real. Cabe mencionar, que al igual que nuestro simulador MSS[1] [10], numerosas alternativas utilizan el motor gráfico Unity 3D, uno de los más potentes, en cuanto a rendimiento y gráficos. En la Tabla 2.2, se comparan algunas de las características más relevantes de los simuladores estudiados con el simulador MSS. Donde se contrastan sus aspectos generales, como los motores gráficos que emplean y la disponibilidad de su licencia y código fuente. Seguidamente, se comparan algunas características en cuanto al manejo de cámaras en sus entornos virtuales, como la posibilidad de añadir múltiples cámaras, si permite configurar o modificar distintos parámetros de ellas (e.g PTZ), y la capacidad de transmitir frames en tiempo real. Finalmente, se contrastan aspectos de sus escenarios sintéticos, como su libre creación y personalización, los tipos de objetos que existen en ellos, y si están basados en datos reales (e.g. imágenes o vídeos).

Tabla 2.2: Tabla comparativa de simuladores de cámaras más relevantes. Clave: PTZ = Pan-Tilt-Zoom (Giro-Inclinación-Zoom), FPS = Frames Per Second (Imágenes por segundo), E = Éstatico, D = Dinámico, PO = Portátil y PE = Periódico.

Simulador	**Motor gráfico**	**Disponibilidad**		**Cámaras**			**Escenarios**		
		Licencia	**Código**	**Múltiples**	**Parámetros modificables**	**Transmisión online**	**Creación**	**Tipos objetos**	**Datos reales**
Virtual Pedestrian	Propio	No	No	No	Ninguno	No	No	E	Si
3DWM	Propio	No	No	Si	Ninguno	Si	No	E, D y PO	Si
Mirror Worlds	Unity y X3D	No	No	Si	-	Si	No	E y D	Si
Virtual KITTI	Unity	No	No	No	Resolución, FPS	No	No	E, D, PO y PE	Si
MSCV	Blender	No	No	No	Ninguno	No	-	E, D y PE	No
SLNCR	Panda	Si	Si	Si	Posición, PTZ	-	Si	E y D	No
SVR	OpenGL	No	No	Si	Posición, PTZ	Si	No	E y D	No
OVVV	Source	Si	No	Si	Posición, PTZ, resolución, FPS	Si	Si	E, D y PO	No
OVVV extensión	Source	Si	Si	Si	Posición, PTZ, resolución, FPS	Si	Si	E, D y PO	No
UnrealCV	UE 4	Si	Si	No	Posición, PTZ	No	Si	E	No
SceneNET RGB-D	UE 4 y Unity	No	No	No	-	No	No	E	No
PHAV	Unity	Si	Si	No	-	No	Si	E, D, PO y PE	No
COnGRATS	Unity	No	No	Si	Posición, PTZ	No	No	E, D, PO y PE	No
Synthia	Unity	Si	No	Si	Posición.	Si	No	E, D, PO y PE	No
MSS	Unity	Si	Si	Si	Posición, PTZ, resolución, FPS	Si	Si	E, D, PO y PE	No

[1] https://repositorio.uam.es/handle/10486/677829

Capítulo 3

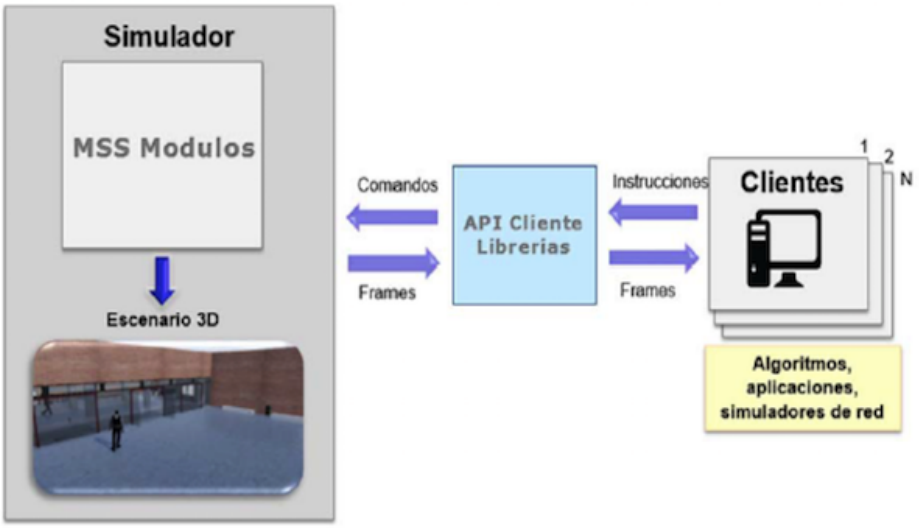
Diseño

Este capítulo muestra el diseño seguido en este proyecto. En la sección 3.1 se analiza el simulador MSS. En la sección 3.2 se estudian escenarios y situaciones de interés. Seguidamente, en la sección 3.3 se presenta una taxonomía de los tipos de objetos existentes en los escenarios, detallando cuales son útiles para definir la extensión y ambientación del escenario, y cuales para proporcionar situaciones de interés. Finalmente, en la sección 3.4 se describen los requisitos que debe cubrir el sistema encargado de gestionar los eventos que realizarán los objetos dinámicos.

3.1. Multi-camera System Simulator (MSS)

Multi-camera System Simulator (MSS) [10], diseñado por Mario González para el VPU-lab, ofrece un entorno virtual de pruebas útil para simular situaciones en diversos entornos y condiciones. Esta herramienta esta basada en una arquitectura Cliente-Servidor (mostrándose su arquitectura software en la Figura 3.1a), y en el motor gráfico Unity 3D, del cual se extiende la funcionalidad con el objetivo de manejar múltiples cámaras distribuidas en tiempo real, pudiendo configurar sus propiedades y retrasmitir frames. Cuenta con dos escenarios base: *EPSlite* y *EPSfull*. El escenario *EPSlite* simula el edificio A-*Alan Turing* de la Escuela Politécnica Superior de la UAM (puede verse en la Figura 3.1b), tiene una extensión de unos 600 metros cuadrados, y está compuesto por objetos estáticos y por una docena de personajes animados obtenidos del Asset Store de Unity[1], y de diversas páginas de modelos 3D[2][3][4]. El escenario *EPSfull*, es una extensión del entorno anterior y cuenta con vehículos.

[1] https://assetstore.unity.com
[2] https://www.cgtrader.com
[3] https://tf3dm.com/
[4] https://www.turbosquid.com



(a)

(b)

Figura 3.1: Simulador MSS: a) Arquitectura software. b) Escenario *EPSlite*. ([10])

3.2. Escenarios y eventos de interés

Como bien sabemos, no todos los escenarios resultan igual de interesantes y de relevantes a la hora de ser estudiados, y es por eso, por lo que es necesario identificar que escenarios pueden ser característicos para ser analizados [11]. Los eventos y las situaciones de interés que ocurren en los escenarios, son provocados por todo tipo de objetos móviles o dinámicos. A continuación se describen escenarios de interés en el ámbito de este TFG (algunos ejemplos se muestran en la Figura 3.2):

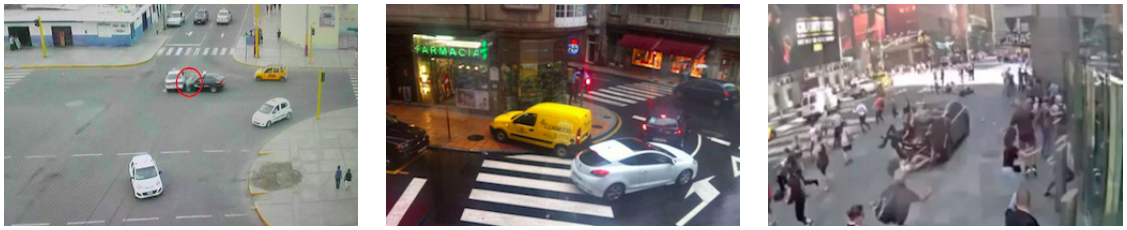
- Monitorización de tráfico en cruces de calles y carreteras.
- Video-seguridad en estaciones de trenes y autobuses, aeropuertos.
- Video-vigilancia en cárceles, manifestaciones y concentraciones.
- Video-seguridad en lugares públicos como parques, plazas y monumentos, hospitales, tiendas y mercados, universidades, conciertos y estadios, etc.
- Eventos deportivos (como el VAR en el futbol).

(a) (b) (c)

Figura 3.2: Ejemplo de escenarios de interés: a) Monitorización de tráfico. (https://goo.gl/9j39qt) b) Videoseguridad en lugares públicos. ([12]) c) Eventos deportivos. (https://goo.gl/2skbQb)

En este Trabajo de Fin de Grado hemos decidido centrarnos en un escenario de una ciudad con diversos cruces de calles, en el cual sucedan diversas situaciones de interés. A continuación se estudian los eventos y situaciones de interés que nos podemos encontrar en este tipo de escenarios, y en la Figura 3.3 se pueden apreciar ejemplos de alguno de ellos:

- Movimiento y coordinación entre personas, vehículos y semáforos.
- Colisiones entre vehículos, personas y objetos.
- Acciones indebidas de las personas y de los vehículos (e.g. personas cruzando indebidamente las calles o vehículos realizando giros donde se permite).
- Vehículos mal estacionados y vehículos que los adelantan o evitan.
- Interacciones de las personas con otros personas y objetos del escenario.



(a) (b) (c)

Figura 3.3: Ejemplo de situaciones de interés: a) Choque entre vehículos. (https://goo.gl/q7ip1e) b) Vehículo mal estacionado. (https://goo.gl/Uz7338) c) Atropellos de personas. (https://goo.gl/nVDERH)

3.3. Tipos de objetos de un escenario

Los objetos del escenario se pueden clasificar en función de su capacidad para controlar el inicio o la finalización de su propio movimiento [13]. En la Figura 3.4 se muestra la taxonomía que hemos seguido para clasificar los objetos de los escenarios. Por lo tanto, los **objetos dinámicos** son objetos que tienen control para empezar o terminar su movimiento por si mismos, en cambio, los **objetos contextuales** son objetos que carecen de control para iniciar o terminar su propio movimiento.

Dependiendo de la naturaleza del movimiento de cada objeto, tenemos tres subclases:

- **Objetos estáticos:** permanecen siempre fijos e inmóviles, no pueden ser desplazados por ningún otro objeto del escenario.
- **Objetos portátiles:** pueden ser desplazados por objetos dinámicos, moviéndose desde y hasta que el objeto dinámico que actúa sobre él lo decida.
- **Objetos periódicos:** se encuentran siempre en continuo movimiento, siendo este de forma periódica ajenos al resto de objetos de la escena.

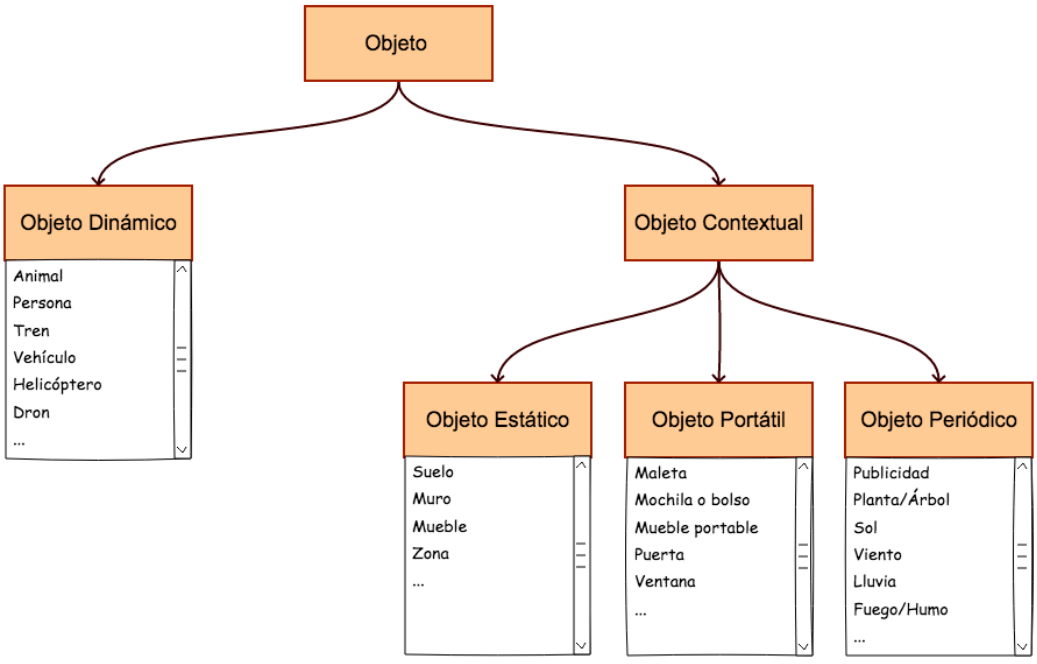


Figura 3.4: Taxonomía de los objetos de un escenario.

Un escenario debe estar formado mínimamente por objetos estáticos (e.g. muros, zonas, suelos, muebles), los cuales definen la extensión y en el ambiente de dicho escenario. Los objetos dinámicos (e.g. personas, animales, trenes, vehículos, drones) y los objetos portátiles (e.g. maletas, mochilas, muebles portables, puertas, ventanas), se encargan de proporcionar situaciones de interés a los escenarios, otorgando mayor complejidad. Finalmente, los objetos periódicos (e.g. publicidades, plantas, partículas), dan mayor realismo a las escenas, generándose escenarios de gran calidad.

3.4. Requisitos del sistema de gestión de objetos

Una vez estudiados las diversas situaciones de interés que pueden suceder en un escenario específico de una ciudad y de haber definido y formalizado los tipos de objetos que pueden existir, se describen las funcionalidades que debe tener un sistema encargado de gestionar y coordinar los objetos dinámicos (vehículos y personas) de un escenario, con el fin, de reproducir situaciones y eventos de interés. Para facilitar la compresión de las funcionalidades de los requisitos del sistema, los diferenciamos en dos categorías: requisitos funcionales y requisitos no funcionales.

3.4.1. Requisitos funcionales

Los requisitos funcionales que se describen a continuación son las funcionalidades que se deben desarrollar para lograr los objetivos de este proyecto:

RF1: Debe proporcionar coordinación entre vehículos, entre personas, entre vehículos y personas, y frente a semáforos.

RF2: Debe proporcionar choques entre vehículos, entre vehículos y personas, y entre vehículos y objetos del entorno.

RF3: Deben existir vehículos mal estacionados en las calles, y vehículos que los eviten y los adelanten.

RF4: Deben existir distintas velocidades de los vehículos, produciéndose adelantamientos en las calles.

RF5: Deben existir vehículos y personas que realicen acciones indebidas, como circular por zonas de peatones, o cruzar indebidamente las calles.

RF6: Debe haber una gran variedad de modelos de vehículos, tener modelos con luces (frontales, freno, intermitentes), y con humo o fuego.

RF7: Debe haber una gran variedad de modelos de personas y las cuales deben realizar acciones básicas como andar o correr, e interacciones como pelear.

RF8: Debe poderse modificar las densidades de vehículos y personas.

RF9: Las rutas de los vehículos y las personas deben ser aleatorias, así como, la disposición de los mismos sobre el escenario.

3.4.2. Requisitos no funcionales

Para desarrollar un escenario para un simulador apropiado para ser utilizado investigación, se deben considerar estos requisitos no funcionales:

RNF1: Rendimiento. El rendimiento del sistema es crítico. Debe diseñarse para trabajar y retransmitir vídeo a los clientes en tiempo real.

RNF2: Fiabilidad. El sistema debe producir situaciones que se repitan periódicamente para proporcionar una ejecución duradera y continuada.

RNF3: Disponibilidad. El sistema debe poder operar durante largos períodos de tiempo sin que se colapse el tráfico.

RNF4: Documentación. Se debe incluir una documentación adicional, además de tutoriales sobre su uso en el escenario.

3.4.3. Sistemas de tráfico

En el estudio realizado sobre los recursos disponibles del Asset Store de Unity del apéndice C, pueden verse varios sistemas de tráfico de utilidad en escenarios que simulan ciudades. Los recursos *Urban Traffic System Pro* y *Population System Pro* son bastante potentes y cuentan con gran número de modelos de vehículos y personas, pero su gran inconveniente es que no cuenta con una IA de frenado. El recurso *Simple Waypoint System* (SWS), utilizado en los escenarios base *EPSlite* y *EPSfull* del simulador MSS [10], útil para trabajar con objetos dinámicos individuales, no para sistemas de objetos, teniéndose que programar la coordinación entre ellos.

Finalmente, nos decantamos por la elección del recurso *Intelligent Traffic System* (ITS)[5] por su simplicidad y facilidad de uso, y por contar con las funcionalidades necesarias para la implementación de diversos eventos de interés: Cuenta con una IA de frenado al estar frente a vehículos o cruces, permite coordinación entre vehículos y personas, soporta coordinación con semáforos, es totalmente aleatorio, permite modificar la cantidad máxima de objetos que generará y gestionará, cuenta con herramientas de edición de rutas y cruces fáciles de usar, se pueden integrar nuevos modelos de vehículos y personas en el sistema, se pueden configurar y programar diversas situaciones de interés, y permite guardar y cargar rutas creadas.

[5] https://assetstore.unity.com/packages/templates/systems/its-intelligent-traffic-system-source-22765

Capítulo 4

Implementación

Este capítulo describe la implementación llevada a cabo en este proyecto. En la sección 4.1 se describe nuestro punto de partida. En la sección 4.2 se explican los elementos necesarios de cada tipo de objeto de un escenario. Finalmente, en la sección 4.3 se detalla la integración y alteración del sistema de gestión de objetos dinámicos.

4.1. Punto de partida

4.1.1. Escenario virtual: Ciudad

Una vez realizado el estudio sobre los escenarios de interés en la sección 3.2, se realizó un extenso estudio sobre los recursos disponibles del Asset Store de Unity, el cual puede verse en el apéndice C. Para el desarrollo de un escenario en el cual se simule una ciudad, hemos decidido partir de un escenario completo *Modern City Pack*[1]. Es un escenario de alta calidad, con numerosos objetos estáticos y periódicos, detalles y texturas, al cual se le pretenden añadir objetos dinámicos, encargados de proporcionar eventos y situaciones de interés Además cuenta con dos versiones del mismo escenario, de día y de noche. La extensión aproximada del escenario es de unos 65.000 m2. En la Figura 4.1 se puede observar el escenario *Modern City Pack*.

Figura 4.1: Imágenes del escenario *Modern City Pack* (https://goo.gl/trenoj).

[1] https://assetstore.unity.com/packages/3d/environments/urban/modern-city-pack-18005

4.1.2. Modificaciones y ajustes del escenario

Antes de implementar el sistema encargado de gestionar los objetos dinámicos, se tuvieron que editar las dos escenas, (una que agrupaba objetos con texturas diurnas y otra con texturas nocturnas), que incluía el paquete descargado. Esto se debía, al posicionamiento de los objetos de las escenas, los cuales dejaban zonas del escenario sin definir, como puede verse en la Figura 4.2a . Por lo que se posicionaron y crearon minuciosamente diversos objetos estáticos, con el fin de cerrar el escenario y crear rutas cíclicas sobre las calles, tal y como puede verse en la Figura 4.2b.

(a) (b)

Figura 4.2: Comparativa: a) Escenario original. b) Escenario modificado.

4.2. Implementación de objetos

En esta sección se explican los elementos que puede tener todo tipo de objeto existente en un escenario (basándonos en manuales de creación de objetos en Unity 3D [14, 15]), siguiendo la taxonomía de tipos de objetos definida en la etapa de diseño y estudiada en este documento en la sección 3.3. Para mayor información sobre la creación de cada tipo de objeto véase el apéndice D.

4.2.1. Elementos de un objeto estático

Un objeto estático es aquel que permanece siempre fijo e inmóvil, y nunca puede ser desplazado. Esta clase de objeto es la que se necesita mínimamente para la creación de un escenario, puesto que ellos son los que definen la extensión y ambientación. En la Figura 4.3 se muestra un diagrama de los elementos básicos de un objeto estático.

Los objetos estáticos se componen necesariamente por los siguientes elementos:

- **Posicionamiento:** define la posición, rotación y escala en el espacio 3D. Son características que definen las coordenadas donde se encuentra el objeto, su orientación en el espacio, y el tamaño del mismo.
- **Modelo geométrico:** se encarga de darle una forma al objeto, ya sea simple como un cubo o complejo como un edificio.

- **Materiales y texturas:** se encargan de rellenar y pintar dicho modelo geométrico del objeto, ya sea únicamente añadir un material de ladrillos a un muro, o varios materiales de aceras, calzadas y bordillos a una calle.

- **Área de colisiones:** define el área o volumen de un objeto para los propósitos de colisiones físicas. Normalmente este elemento suele tener el modelo geométrico del objeto. Si el modelo geométrico es muy complejo, se puede añadir uno similar, más sencillo, que aporte un rendimiento adecuado sin sobrecargar computacionalmente el procesador al detectar una colisión con dicho objeto.

- **Sombras y reflejos:** guardan gran relación con el modelo geométrico del objeto y con las propiedades del material o textura que se le añadan al objeto.

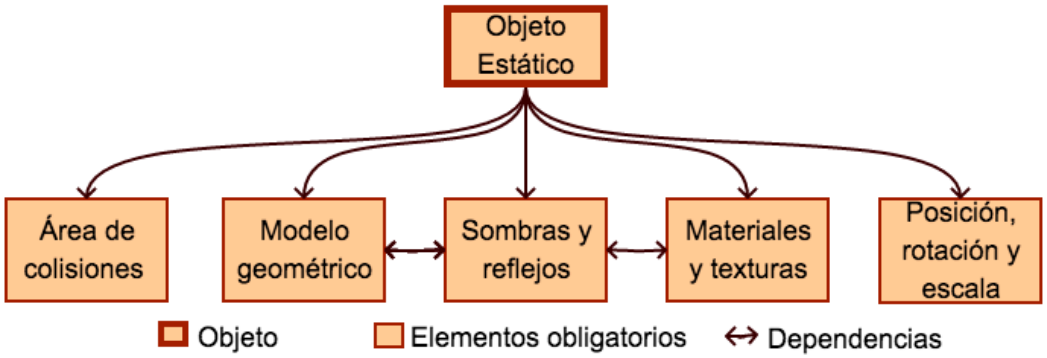


Figura 4.3: Diagrama de bloques de los elementos que componen un objeto estático.

En la Figura 4.4 puede verse un ejemplo de un objeto estático (Calle) implementado en Unity. En cuanto a características o elementos opcionales que pueden tener esta clase de objetos nos podemos encontrar los siguientes:

- No todos los objetos estáticos tienen la misma complejidad, ya que se pueden crear objetos simples como un muro, con un material de ladrillo, un área de colisiones y un modelo geométrico, o crear un muro complejo, compuesto por varios objetos estáticos (pared, ventanas, puerta, escaleras).

- En algunos casos, el conjunto de objetos que componen un objeto estático no necesitan tener el componente de área de colisiones, si por ejemplo un objeto estático con un modelo geométrico simula una puerta estática, la cual forma parte de un objeto muro (siendo este el objeto padre) el cual ya tiene definidas dichas áreas de colisiones, no sería necesario atribuírselas al objeto hijo.

- Los objetos estáticos pueden estar formados por dos o más objetos que representen el mismo objeto pero con distintas calidades (en Unity se llama LOD: *Level of Detail*), lo que permite reducir el número de triángulos renderizados para un objeto a medida que la distancia desde él a la cámara aumente, reduciendo así, la carga en el hardware y mejorando el rendimiento de renderización.

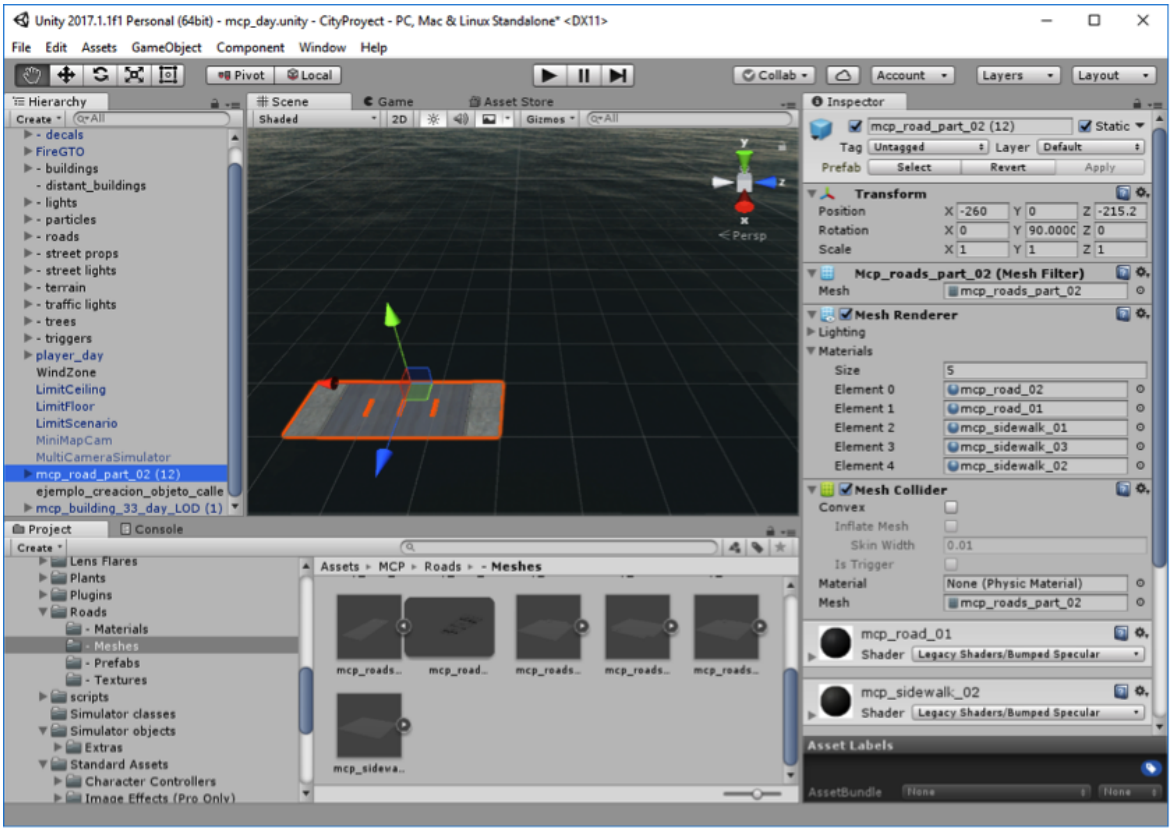


Figura 4.4: Ejemplo de objeto estático implementado en Unity.

4.2.2. Elementos de un objeto portátil

Un objeto portátil es un objeto que por sí mismo no puede desplazarse ni realizar ningún tipo de modificación en cuanto a su posición o rotación, pero que puede ser desplazado o forzado a modificar su posición o rotación, siempre y cuando, interactúe sobre el un objeto dinámico. Esta clase de objeto se encarga de proporcionar eventos y situaciones de interés a los escenarios, otorgándoles mayor complejidad y calidad.

Los objetos portátiles extienden a los objetos estáticos tal y como puede verse en la Figura 4.5, donde se muestra un diagrama de los elementos básicos de un objeto portátil y los elementos del objeto base (objeto estático). En comparación con un objeto estático, existe un nuevo elemento: **Script o controlador**. Éste permite a un objeto dinámico relacionarse con el objeto portátil, el cual detecte y controle cuando un objeto dinámico ha interactuado con él, para iniciar o terminar su modificación de posición. Normalmente esta funcionalidad se suele controlar y gestionar con *Scripts*. La posición, rotación y escala guardan gran relación con el controlador o *Script* del objeto.

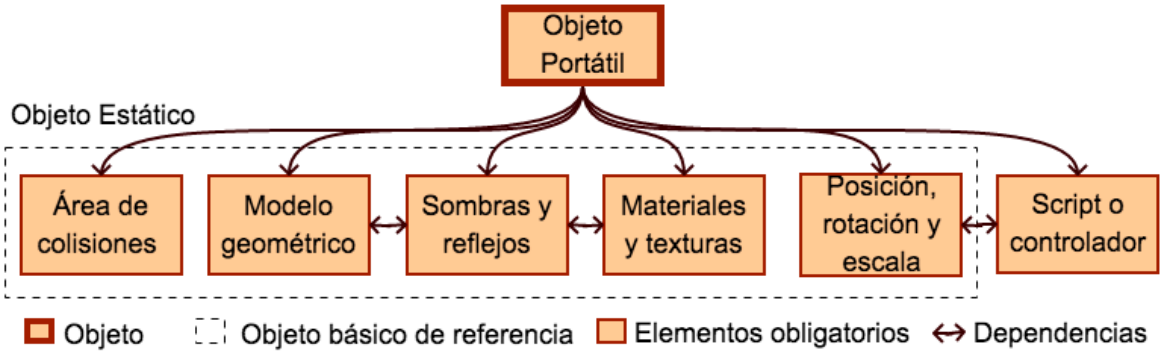


Figura 4.5: Diagrama de bloques de los elementos que componen un objeto portátil.

Los objetos portátiles pueden tener los elementos opcionales que tienen los objetos estáticos, explicados anteriormente. En la Figura 4.6 puede verse un ejemplo de un objeto portátil (Puerta abatible) implementado en Unity.

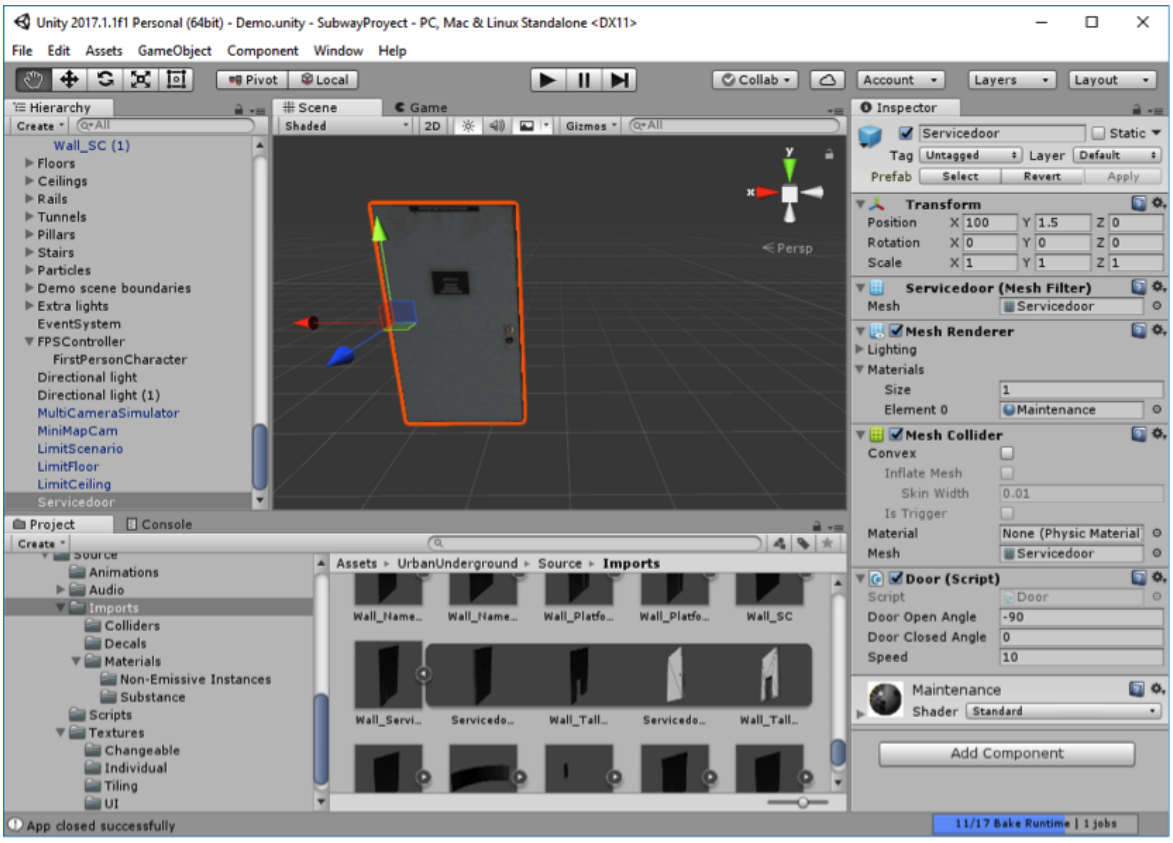


Figura 4.6: Ejemplo de objeto portátil implementado en Unity.

4.2.3. Elementos de un objeto dinámico

Un objeto dinámico es un objeto que por sí mismo puede desplazarse y por lo tanto realizar cualquier modificación de su posición o rotación, es decir, son objetos que tienen un control sobre el comienzo y la finalización de su propio movimiento a lo largo de una trayectoria, ruta o recorrido definido previamente. Esta clase de objeto permite definir eventos y situaciones de interés en los escenarios, otorgándoles mayor complejidad.

Los objetos dinámicos extienden a los objetos estáticos tal y como puede verse en la Figura 4.7, donde se muestra un diagrama de los elementos básicos de un objeto dinámico y los elementos del objeto base (objeto estático).

- **Script o controlador:** permite al objeto dinámico iniciar o terminar su modificación de posición o rotación. Normalmente esta funcionalidad se suele controlar y gestionar con *Scripts*. La posición, rotación y escala guardan gran relación con el controlador o *Script* del movimiento del objeto.

- **Ruta o recorrido:** seguida por el objeto dinámico, desplazándose así por todos los puntos del recorrido que lo conformen, rotándose en el momento necesario para llegar al siguiente punto del recorrido. Normalmente esta funcionalidad se suele controlar y gestionar con *Scripts*. El *Script* del movimiento del objeto dinámico guarda gran relación con el controlador o *Script* de la ruta del objeto.

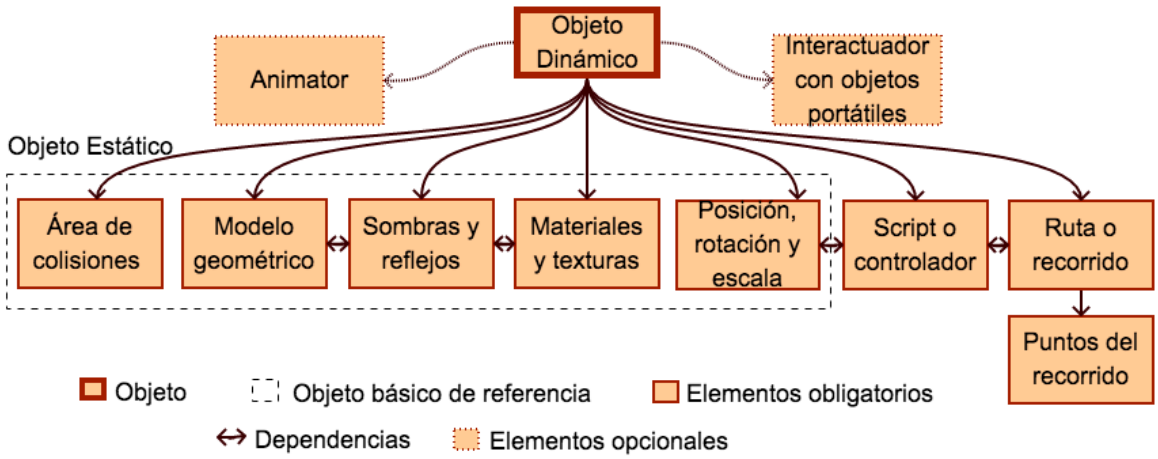


Figura 4.7: Diagrama de bloques de los elementos que componen un objeto dinámico.

En la Figura 4.8 puede verse un ejemplo de un objeto dinámico (Vehículo) implementado en Unity. Los objetos dinámicos pueden tener los elementos opcionales que tienen los objetos estáticos, y además contar con los siguientes:

- Objetos dinámicos como personas o animales necesitan además tener una animación, normalmente controlada con un **Animator,** que indique una transición de estados y de distintos movimientos que realizaría al moverse, pararse, correr, etc. Este componente otorga naturalidad al objeto dinámico.

- Algunos objetos dinámicos necesitaran un **Script interactuador con objetos portátiles** encargado de relacionarse con los *Scripts* de los objetos portátiles, con el fin de modificar la posición o rotación del objeto portátil.

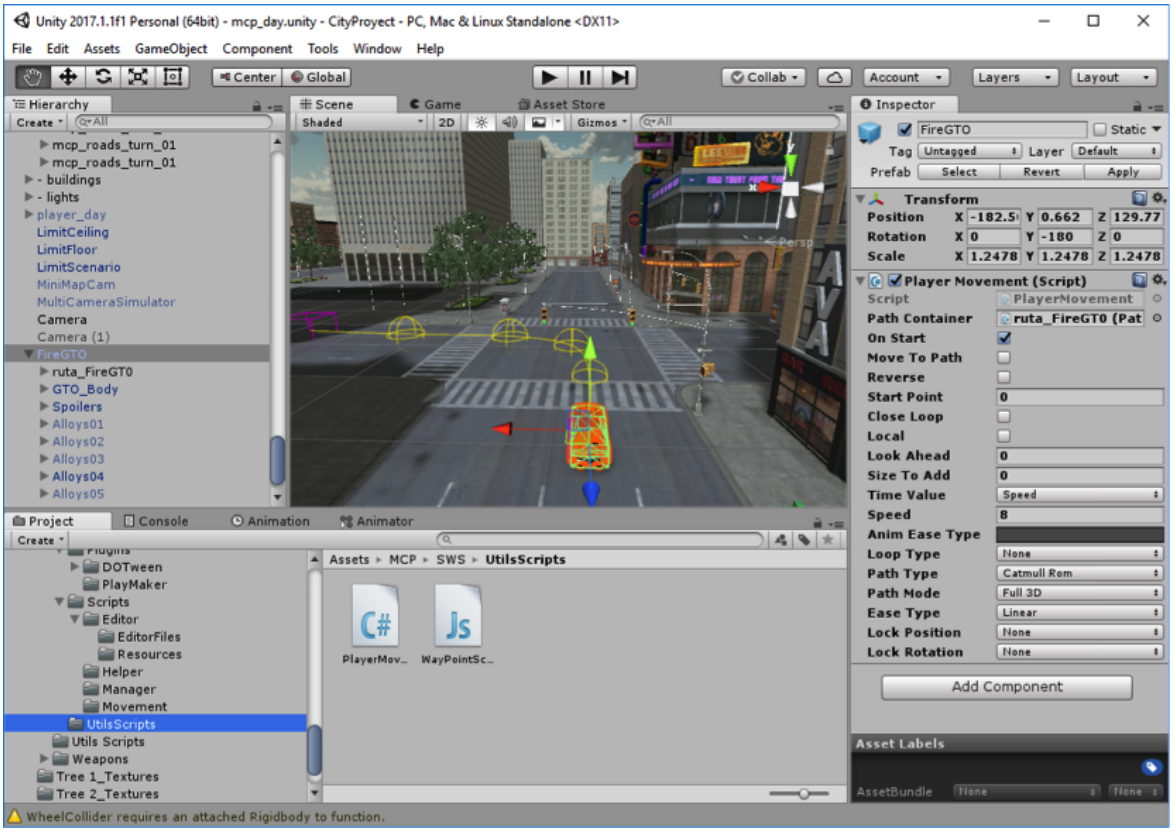


Figura 4.8: Ejemplo de objeto dinámico implementado en Unity.

4.2.4. Elementos de un objeto periódico

Un objeto periódico es un objeto que por sí mismo puede modificar su posición, rotación o escala, pero de forma periódica y sin tener cierto control sobre dicho movimiento o modificación de propiedades, no pudiendo decidir en ningún caso cuando parar, estando siempre en continuo movimiento. Esta clase de objeto da mayor realismo a las escenas, generando escenarios de gran calidad.

Para clasificar este tipo de objetos se han diferenciado en si son corpóreos o no, es decir, si son objetos con un modelo geométrico definido, dándole una forma a dicho objeto, o si son objetos que carecen de un modelo geométrico que los defina.

Los objetos periódicos corpóreos extienden a los objetos estáticos tal y como puede verse en la Figura 4.9, donde se muestra un diagrama de los elementos básicos de un objeto periódico corpóreo y los elementos del objeto base (objeto estático).

- **Controlador:** permite al objeto periódico modificar de posición o rotación constantemente. Esta funcionalidad se suele controlar y gestionar de tres formas diferentes: mediante *Scripts*, mediante animaciones con *Animation* o *Animator*, o mediante un controlador de agente externo que interactúe con el objeto externo *Wind Zone*, que se encuentra en la propia herramienta de edición de un objeto árbol. La posición, rotación y escala guardan gran relación con el controlador del movimiento del objeto.

Los objetos periódicos incorpóreos se componen por los siguientes elementos:

- **Posicionamiento:** define la posición, rotación y escala en el espacio 3D. Son características que definen las coordenadas donde se encuentra el objeto, su orientación en el espacio, y el tamaño del mismo.

- **Materiales y texturas:** se encargan de rellenar y pintar las partículas que conforman el sistema.

- **Sistema de partículas:** cuenta con diversas propiedades configurables, ajustándose a las necesidades del objeto final que desee crear el usuario. Este elemento guarda gran relación con la posición, rotación y escala del objeto y con las propiedades del material o textura que se le añadan al objeto.

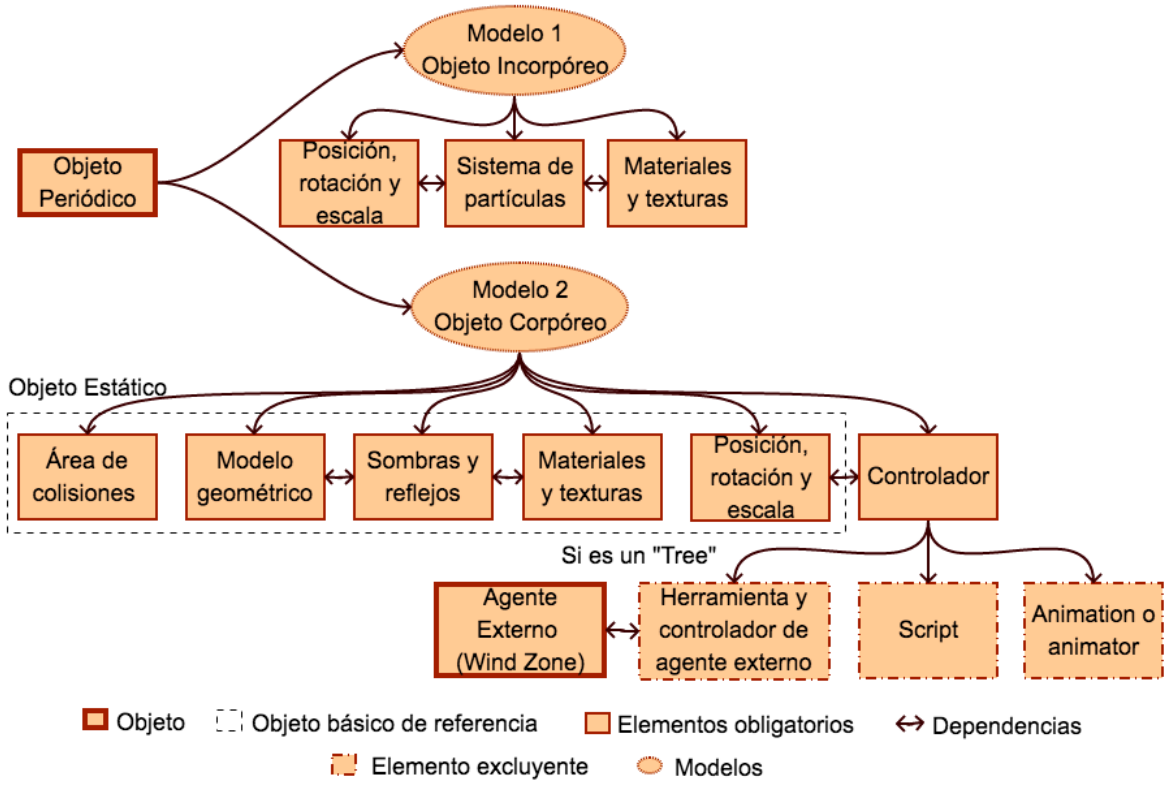


Figura 4.9: Diagrama de bloques de los elementos que componen un objeto periódico.

Los objetos periódicos corpóreos pueden tener los elementos opcionales que tienen los objetos estáticos, explicados anteriormente. En la Figura 4.10 puede verse un ejemplo de un objeto dinámico (Árbol) implementado en Unity.

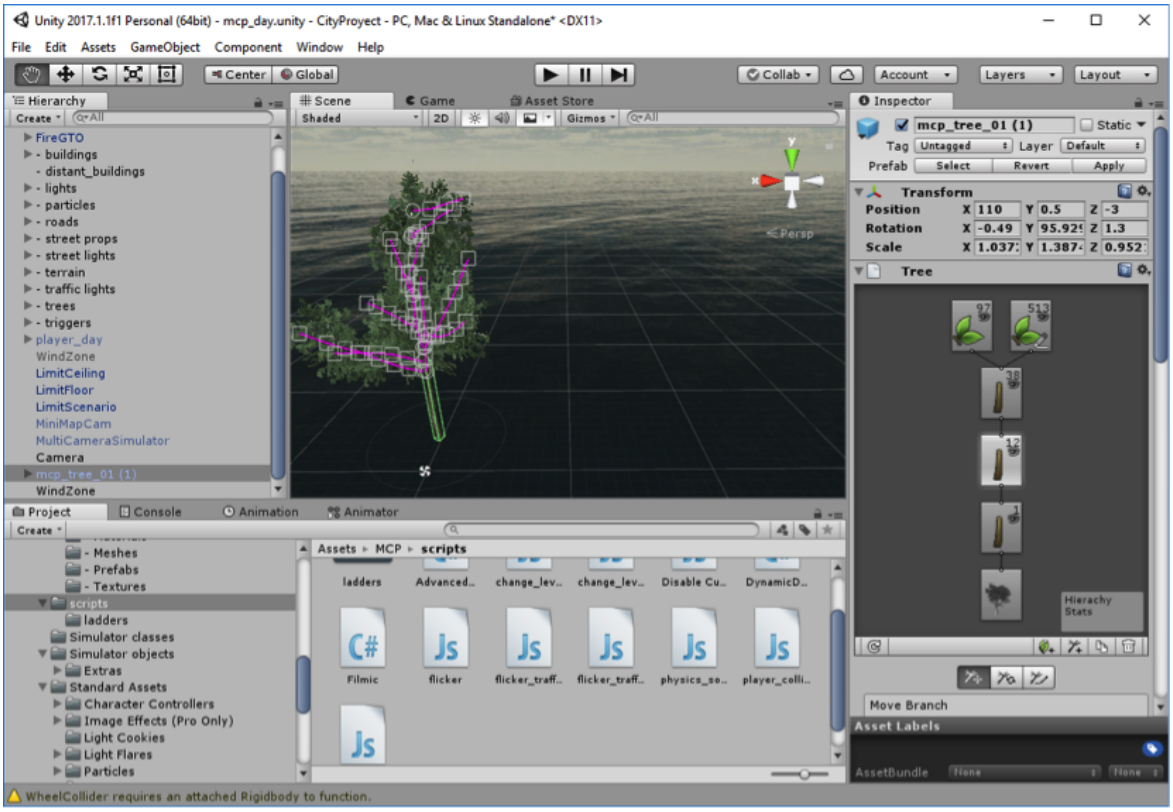
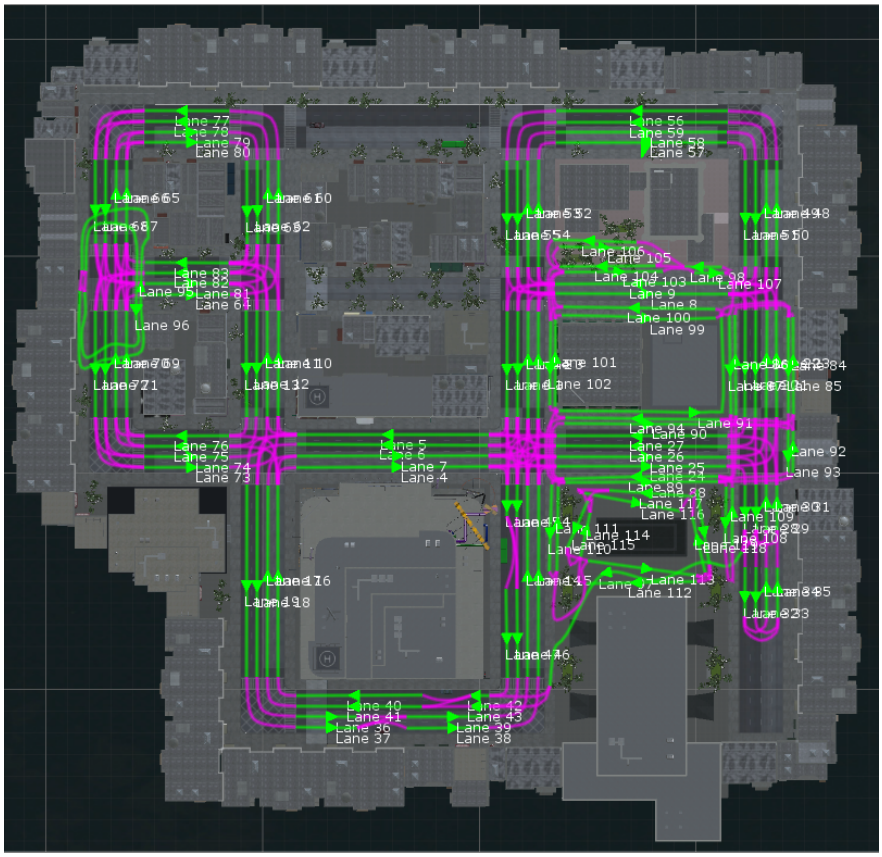


Figura 4.10: Ejemplo de objeto periódico implementado en Unity.

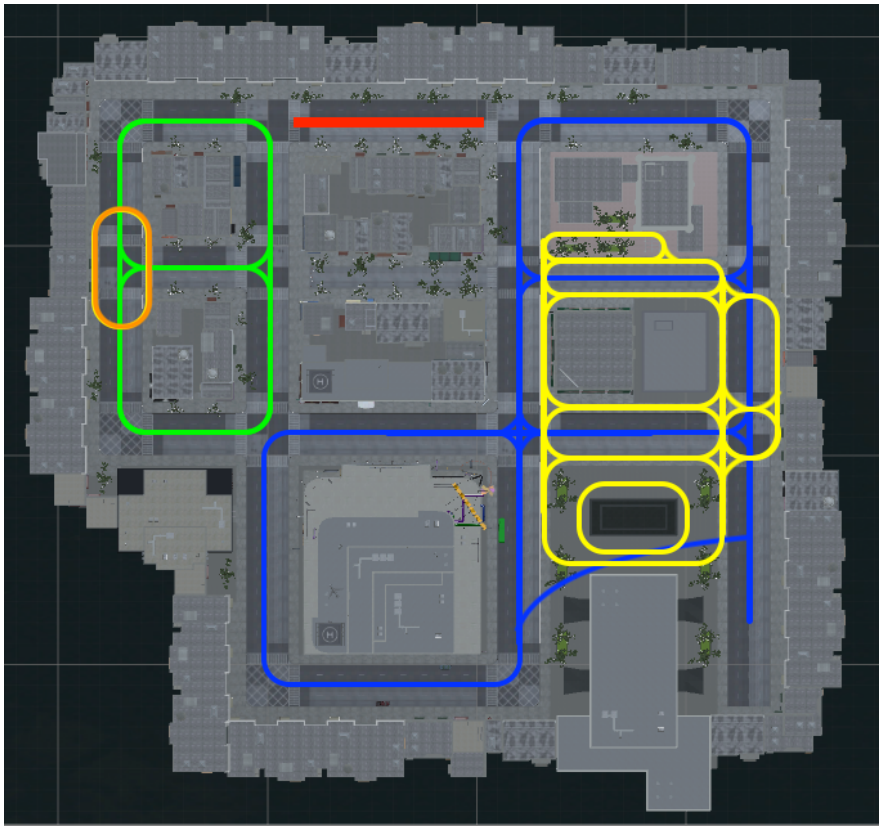
4.3. Implementación del sistema de gestión de objetos

Una vez descargado el sistema de tráfico, se integra y ajusta a nuestro escenario, tal y como puede verse en la Figura 4.11a, donde las líneas rosas representan los cruces y las zonas donde los vehículos y personas comprueban si tienen prioridad para seguir circulando, o en cambio deben ceder el paso. En cambio las líneas verdes representan zonas donde los vehículos o personas circulan sin tener en cuenta el entorno. Cabe mencionar, que tanto el número y los modelos de vehículos y personas, como el posicionamiento y las rutas que toman son totalmente aleatorias, pudiendo definir únicamente el máximo de objetos dinámicos a generar en el escenario. Esta característica del sistema de tráfico resulta realmente interesante, puesto que puede proporcionar vídeos y situaciones realmente diferentes en cada ejecución.

Tras integrar el sistema de tráfico en el escenario, se crean dos instancias de generadores (*Spawner*) de objetos dinámicos, uno encargado de generar vehículos, y otro de generar personas. En la Figura 4.11b se puede ver un esquema del sistema de tráfico completo. En el circuito azul y verde, circulan vehículos, siendo independientes, ya que el circuito verde puede tender al colapso por las colisiones de los vehículos, permitiendo así, que el circuito azul siga ejecutándose sin problemas. En los circuitos amarillo y naranja circulan personas, pero de nuevo, son independientes, y el circuito naranja, al igual que el verde, puede colapsarse. Cabe mencionar, que debido a la gran complejidad de físicas con la que cuentan los vehículos del sistema de tráfico, no es posible añadirles una física de deformaciones, añadiéndolas en un circuito ajeno al sistema de tráfico (circuito rojo), con dos vehículos con una IA sencilla.



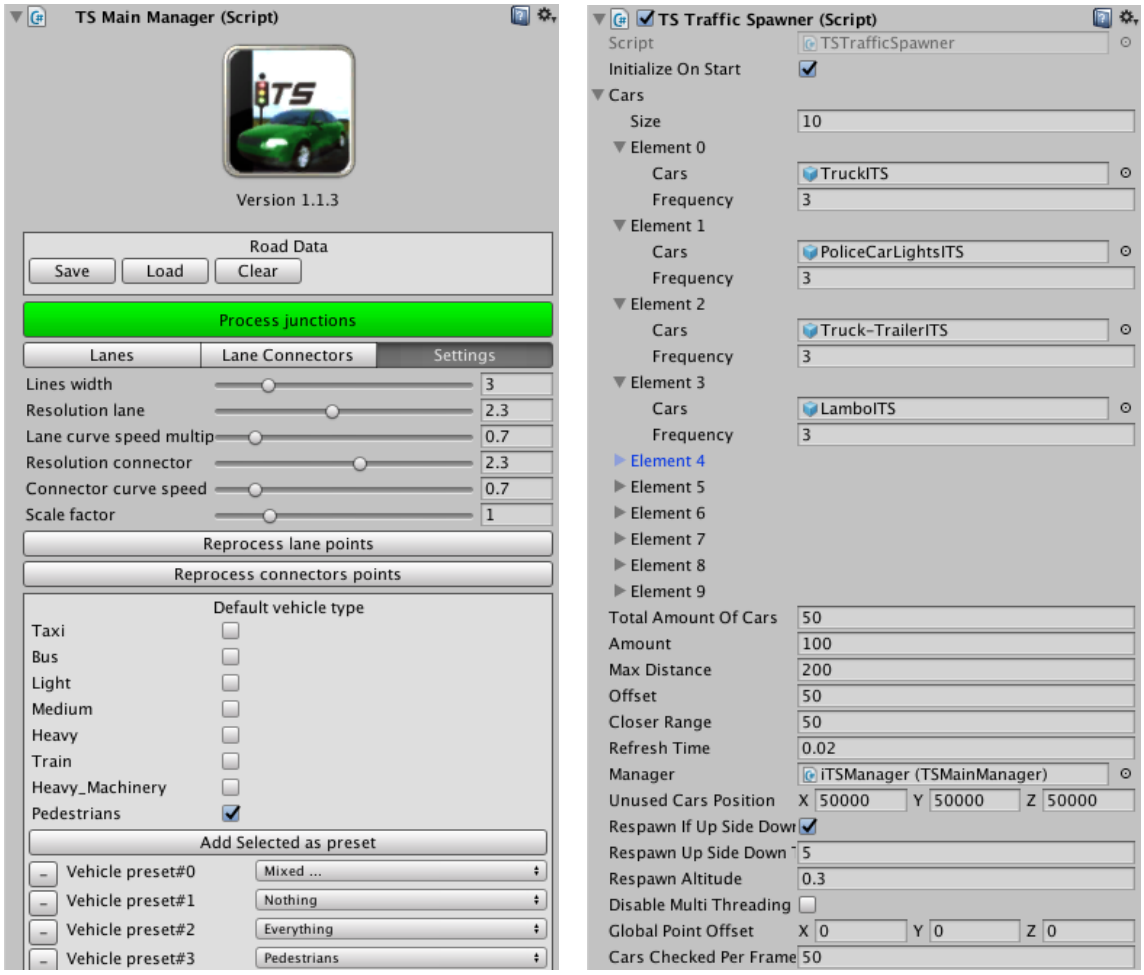
(a)



(b)

Figura 4.11: Implementación del sistema de tráfico: a) Real. b) Esquema.

Para que el sistema de tráfico funcione correctamente, debe existir un objeto especial que cargue las rutas y cruces, y maneje el tráfico de vehículos y personas (Figura 4.12a), y un objeto encargado de generar y posicionar los objetos dinámicos en tiempo de ejecución llamado *Spawner* (Figura 4.12b). Opcionalmente, se pueden incluir varios generadores de objetos para controlar individualmente su cantidad en el escenario, y objetos encargados de proporcionar coordinación con semáforos.



(a) (b)

Figura 4.12: Objetos del sistema de tráfico: a) Manejador de tráfico. b) Generador de vehículos y personas.

El sistema de tráfico proporciona coordinación entre vehículos, personas y semáforos, pero había que trabajar más a fondo, y sacar mayor partido a estas funcionalidades, de tal forma, que se obtuviesen numerosos eventos y situaciones de interés. Uno de los eventos que más nos interesaba tener en el escenario, eran colisiones entre vehículos y atropellos, por lo que se descoordina el sistema de tráfico en un cruce concreto, conseguimos producir accidentes aleatorios. Obteniéndose, como efecto secundario, eventos de interés en el que los vehículos circulan por aceras y zonas de peatones y realizan giros indebidos, o donde los peatones cruzan indebidamente las calles. Colocando vehículos mal estacionados, y programando el sistema de tráfico para que los vehículos detecten cuando deben evitarlos o adelantarlos.

Es importante tener diversos modelos 3D de vehículos (e.g. luces frontales, freno o intermitentes, con humo o fuego) y de personas integrados en el sistema de tráfico, con el objetivo de diferenciarlos. Además es esencial tener a personas andando o corriendo, o vehículos que circulan a distintas velocidades.

Ajeno al sistema de gestión de objetos, pero que era indispensable introducir en el escenario era tener aspectos globales (e.g. lluvia, niebla, viento, sol, día, noche) que aportaran distintas condiciones a los vídeos, modificando luces y sombras como provoca el movimiento del sol, o distorsión en la imagen como provoca la lluvia o niebla. Cabe añadir, que realizar interacciones entre los personajes de un escenario es realmente complicado y costoso, algo, que en el Asset Store de Unity, es dificil de conseguir gratis, por lo que optamos por introducir a dos personas que pelean sencillamente, a partir de las animaciones descargadas en uno de los recursos[2]. Finalmente, se añade un un helicóptero, con una ruta definida[3] sobrevolando las calles del escenario, con una luz que enfocando al suelo, dando cierta complejidad al escenario, y haciendo de él, un escenario de gran calidad y rico en eventos de interés.

Para más información, puede ver el vídeo: *Demo scenario MSS: city*, visitando el link de YouTube[4] donde se detalla la funcionalidad final del escenario descrita anteriormente. En la Figura 4.13 se pueden ver imágenes de ejemplo del vídeo.

(a) (b)

Figura 4.13: Imágenes obtenidas del vídeo: *Demo scenario MSS: city.*

[2] https://assetstore.unity.com/packages/3d/characters/sci-fi-officer-captain-52376
[3] Empleando el recurso Simple Waypoint System (SWS)
[4] https://www.youtube.com/watch?v=BiT-JqPQTaQ

Capítulo 5

Evaluación

Este capítulo presenta los experimentos realizados para evaluar el escenario diseñado e implementado para el simulador MSS descrito en los capítulos 3 y 4.

5.1. Entorno de experimentación

En todos los experimentos usamos una versión compilada con el escenario *EPSlite*, como referencia o base, y cuatro versiones compiladas del escenario de la ciudad, tres de ellas con el escenario diurno con distintas densidades de objetos dinámicos (baja, media y alta), y otra el escenario nocturno. Estas versiones son aplicaciones se ejecutan en un equipo con Windows 10 de 64 bits con las siguientes especificaciones: Intel Core I5-3330 @ 3.00 GHz (4 núcleos y 8 GB RAM), y con una tarjeta gráfica sencilla: Intel HD Graphics. Para cada uno de los experimentos, a cada instancia compilada se conecta un cliente que configura las condiciones del experimento. Cada experimento fue ejecutado por un periodo de tiempo de unos 5 minutos, obteniendo datos cada medio segundo, con el propósito de recopilar los datos necesarios para gráficos (en total 600 datos por cada configuración de cada experimento).

En cuanto al proceso de captura de datos se llevo a cabo de la siguiente manera: Ejecutando la aplicación MSI Afterburner, con la cual recopilamos los datos sobre los recursos, y lanzando la versión compilada del escenario a evaluar, en el servidor, y lanzando los correspondientes programas de pruebas en los clientes. Una vez recopilados y guardados los datos, se llevaban a excel para prepararlos y tratarlos, de tal forma que sean legibles por Matlab, herramienta desde la cual realizamos las gráficas y figuras, representando el comportamiento de recursos frente a las distintas configuraciones del experimento. En la Tabla 5.1 se muestran las herramientas que se han utilizado en los experimentos:

Tabla 5.1: Herramientas empleadas para la realización de los experimentos.

Herramienta	**Desarrollador**	**Propósito**
MSI Afterburner	MSI	Recopilar datos (GPU, CPU y FPS) en cada experimento
Excel	Microsoft	Preparar los datos recopilados para Matlab
Matlab	Mathworks	Generar figuras y calcular distintos estadísticos de los datos

5.2. Resultados experimentales

A lo largo de esta sección, se pretende evaluar el rendimiento de los recursos del equipo en diferentes condiciones. Principalmente se estudian el uso de la CPU y de la GPU, y los FPS que se retransmiten del servidor al cliente, puesto que otros recursos como la RAM o escritura en disco no aportaban información de utilidad, puesto que se mantenían invariables en cada configuración de cada experimento. Se llevarán a cabo varios experimentos, en los que se realizarán diversas configuraciones sobre el escenario Ciudad, realizado y presentado en las secciones anteriores, frente al escenario *EPSlite*, con el fin de poner a prueba dichos recursos:

1. En el **primer experimento** se pretende variar las **densidades de los objetos dinámicos** (de vehículos y personas de escenarios).

2. En el **segundo experimento** se propone variar la **calidad gráfica de los escenarios**, configurándola al ejecutar la versión compilada del escenario.

3. En el **tercer experimento** se pretende variar la **iluminación del escenario**, evaluando una versión **diurna** del escenario frente a otra **nocturna.**

4. En el **cuarto experimento** se propone variar los **FPS a ser retransmitidos al cliente**, configurando los FPS solicitados al ejecutar cada cliente.

5. En el **quinto experimento** se pretende **variar el número de cámaras que retransmiten simultáneamente en cada escenario**, lanzándose tantos clientes como número de cámaras se quiera tener.

En cada experimento, se varían las condiciones propias a dicha evaluación, dejando las demás de forma estándar, tal y como se muestra en la Tabla 5.2:

Tabla 5.2: Condiciones estándar para los experimentos.

Densidad	Calidad	Escenario	Framerate	Cámaras	Resolución
Media	Simple	Día	10 FPS	1 cámara	640x480

5.2.1. Variaciones de densidades de objetos dinámicos

En esta evaluación se parte de una versión compilada del escenario *EPSlite*, y de tres versiones compiladas del escenario de la ciudad, cada una con una densidad distinta. En la Tabla 5.3 se pueden ver las tres configuraciones de densidades que se realizaron en este experimento y en la Figura 5.1 pueden verse ejemplos de densidades de objetos dinámicos.

Tabla 5.3: Distintas configuraciones de densidades de objetos dinámicos.

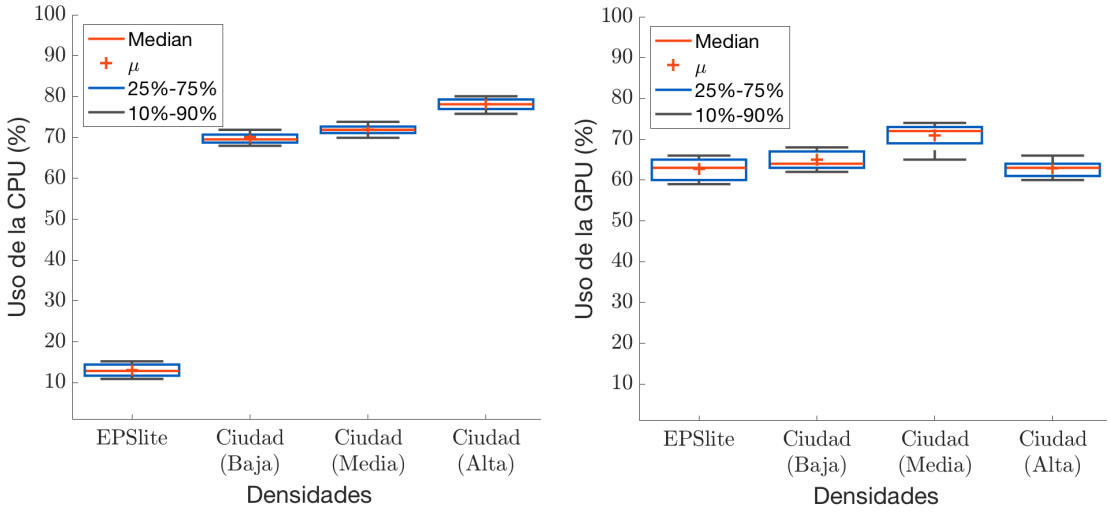
Configuración	Densidad	Máximo de vehículos	Máximo de personas
1	Baja	20	50
2	Media	50	100
3	Alta	200	500

(a) Densidad baja (b) Densidad media (c) Densidad alta

Figura 5.1: Ejemplo de densidades de objetos dinámicos.

En la Figura 5.2a se puede ver que el uso de la CPU para el escenario *EPSlite* es significativamente menor que el escenario Ciudad para distintas densidades. Esto se debe a que tanto el procesamiento computacional de colisiones y coordinaciones, como el número de objetos en la escena provoca que el uso de la CPU varíe, ya que el escenario *EPSlite* cuenta con un edificio sencillo y con una docena de personas con animaciones, y la versión con menor densidad del escenario Ciudad cuenta con más de 100 edificios y con numerosos vehículos y personas en la escena.

En la Figura 5.2b se puede ver que el uso de la GPU varía en pequeña medida para distintas densidades de objetos dinámicos en la escena, ya que la calidad gráfica y la resolución de ellos se mantiene siempre igual (*Simple* y 640x480).



(a) Uso de la CPU (b) Uso de la GPU

Figura 5.2: Uso de recursos a distintas densidades de objetos dinámicos.

En la Figura 5.3 se puede ver que el framerate varía considerablemente cuando la densidad es alta (7 fps), esto se debe a la gran cantidad de objetos dinámicos en la escena y la cantidad de sombras que estos producen, siendo muy costoso seguir retransmitiendo con los mismos FPS mayor cantidad de movimiento en las escenas.

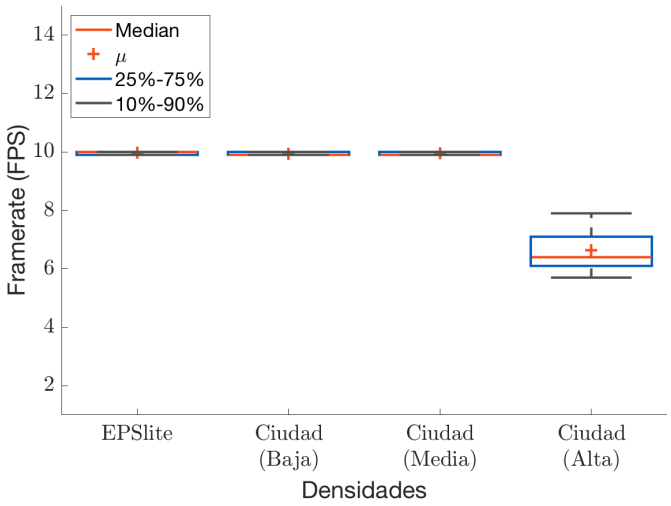


Figura 5.3: Framerate a distintas densidades de objetos dinámicos.

5.2.2. Distintas calidades de los escenarios

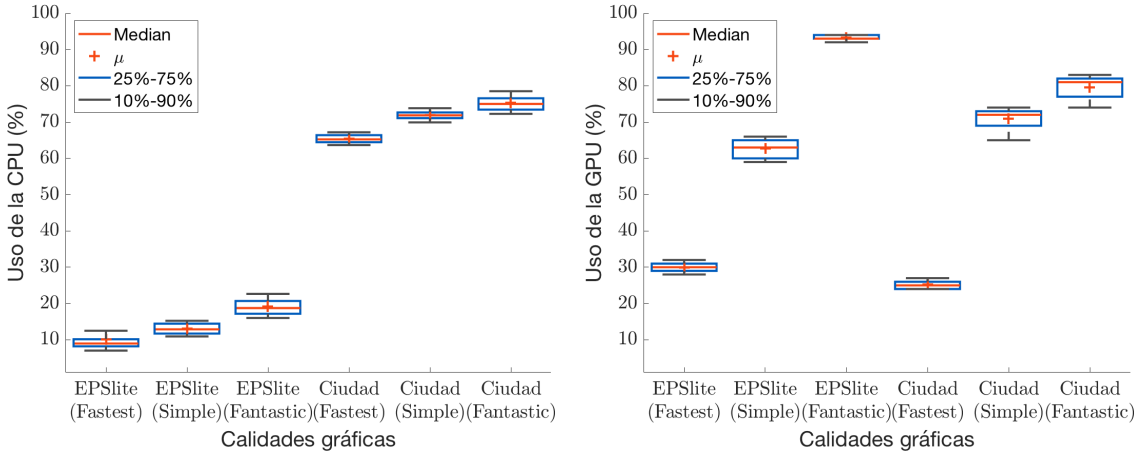
En esta evaluación se parte de una versión compilada del escenario *EPSlite*, y de otra versión compilada del escenario de la ciudad. En la Tabla 5.4 se pueden ver las tres configuraciones de calidad de los escenarios que se realizaron en este experimento y en la Figura 5.4 pueden verse ejemplos de las distintas calidades de los escenarios.

Tabla 5.4: Distintas configuraciones de calidad de los escenarios.

Configuración	**Calidad**	**Detalles**
1	Fastest (Baja)	Sin sombras ni reflejos
2	Simple (Media)	Con sombras y reflejos sencillos
3	Fantastic (Alta)	Con sombras y reflejos detallados

(a) Calidad *Fastest* (b) Calidad *Simple* (c) Calidad *Fantastic*

Figura 5.4: Ejemplo de calidades de los escenarios.



(a) Uso de la CPU (b) Uso de la GPU

Figura 5.5: Uso de recursos a distintas calidades de los escenarios.

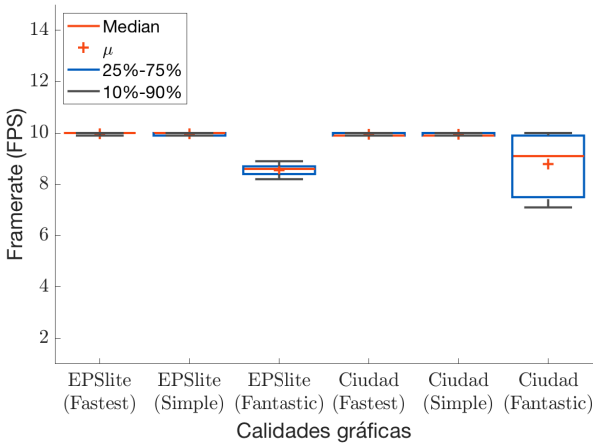


Figura 5.6: Framerate a distintas calidades de los escenarios.

En la Figura 5.5a se puede ver el uso de la CPU para ambos escenarios. Al variar las calidades gráficas no provocan un aumento considerable en el uso de la CPU, puesto que tanto el procesamiento computacional de colisiones y coordinaciones, como el número de objetos en la escena sigue sin alterarse, ya que ambos provocan que el uso de la CPU aumente.

En la Figura 5.5b se puede ver que el uso de la GPU es similar en ambos escenarios, puesto que el renderizado de sombras y reflejos es lo que aumenta en gran medida el uso de la GPU, manteniéndose baja cuando la calidad es *Fast*est, y no hay que generar ni sombras ni reflejos, media para la calidad *Simple*, cuando hay que generar sombras y reflejos sencillos, y alta cuando la calidad es *Fantastic*, cuando se generan sombras y reflejos con gran detalle.

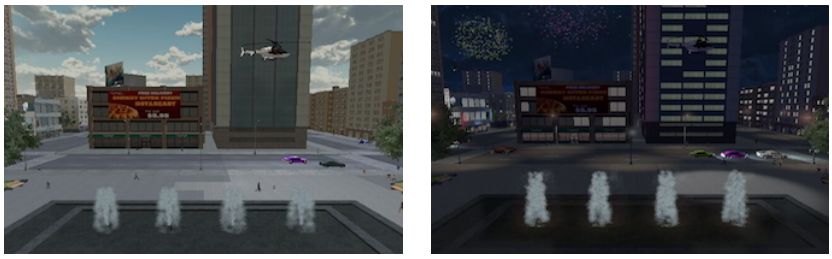
En la Figura 5.6 se observa que el framerate varía para la calidad más alta (*Fantastic*) en ambos escenarios, retransmiten a 8 fps, puesto que renderizan sombras y reflejos con gran detalle. En cambio en calidades media y baja (*Simple* y *Fastest*), se mantiene en 10 fps, por la sencillez de las sombras y reflejos. Cabe mencionar que la gran variación de los FPS en el escenario Ciudad a la calidad más alta es debida la orientación del sol, lo cual genera sombras y reflejos dinámicos, provocando momentos con mayores sombras (menor FPS), y en otros menores sombras (mayor FPS).

5.2.3. Distinta iluminación del escenario

En esta evaluación se parte de una versión compilada del escenario *EPSlite*, y de dos versiones compiladas del escenario de la ciudad: una diurna, y otra nocturna. En la Tabla 5.5 se pueden ver las dos configuraciones de iluminación del escenario que se realizaron en este experimento y en la Figura 5.7 pueden verse ejemplos de iluminación de un mismo escenario.

Tabla 5.5: Distintas configuraciones de iluminación del escenario.

Configuración	**Iluminación**	**Detalles**
1	Diurna	Menor número de luces, sombras y reflejos
2	Nocturna	Mayor número de luces, sombras y reflejos



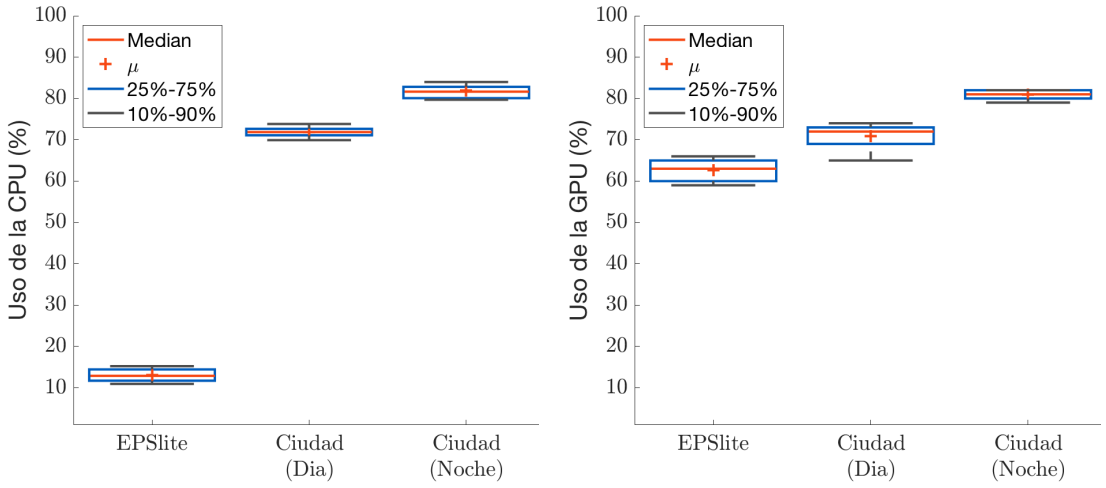
(a) Escenario de día (b) Escenario de noche

Figura 5.7: Ejemplo de iluminación del escenario.

En la Figura 5.8a se puede ver que el uso de la CPU vuelve a ser significativamente menor para el escenario de la *EPSlite*, debido a las pocos objetos que existen en el y las pocas sombras que estos producen, aparte de no contar con materiales que produzcan reflejos. El escenario Ciudad nocturno sobrecarga un 10 % más la CPU, que el escenario Ciudad diurno, por la cantidad de luces independientes que hay en él, las cuales provocan gran cantidad de sombras y reflejos.

En la Figura 5.8b se puede ver que el uso de la GPU varía en pequeña medida para distintas iluminaciones de los escenarios, ya que la calidad gráfica y la resolución de ellos se mantiene siempre igual (*Simple* y 640x480).

En la Figura 5.9 se puede ver que el framerate nos muestra como el escenario de nocturno es el de mayor carga, estando en los 6 fps, frente a los otros dos escenarios se mantienen en los 10 fps. Esto es debido a la gran cantidad de luces en escaparates, publicidades, vehículos y farolas del escenario, las cuales provocan numerosas sombras y reflejos.



(a) Uso de la CPU (b) Uso de la GPU

Figura 5.8: Uso de recursos a distinta iluminación del escenario.

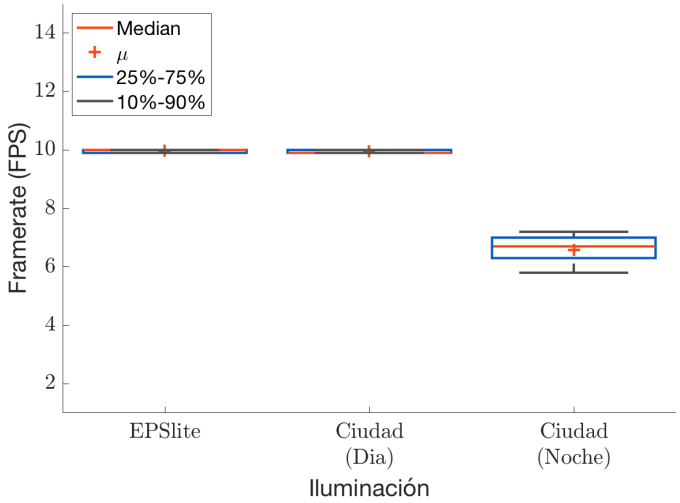


Figura 5.9: Framerate para distinta iluminación del escenario.

5.2.4. Distintos framerate a retransmitir al cliente

En esta evaluación se parte de una versión compilada del escenario *EPSlite*, y otra de la ciudad. Pero se tenían distintos clientes de pruebas, configurados para solicitar FPS diferentes. En la Tabla 5.6 se pueden ver las tres configuraciones de framerate (FPS) a retransmitir a los clientes que se realizaron en este experimento.

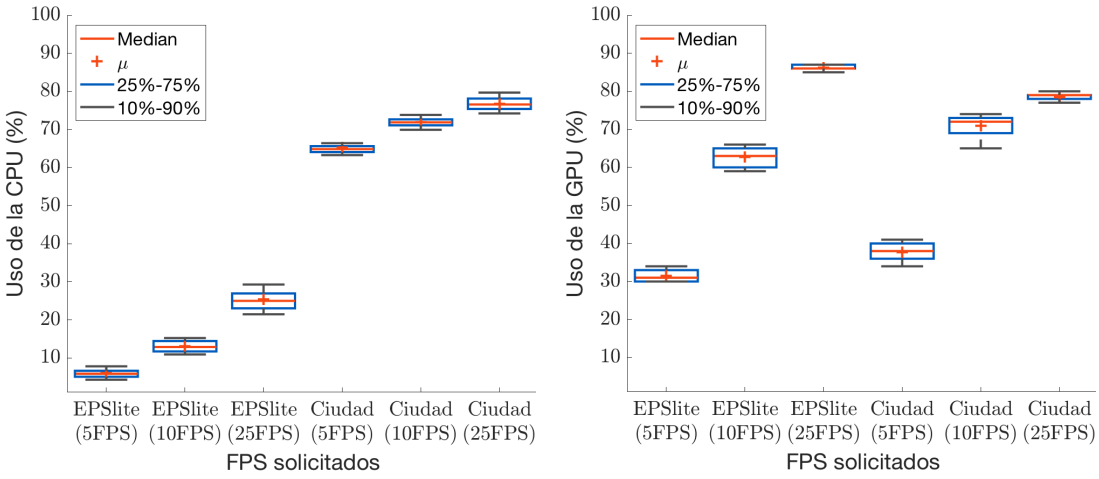
Tabla 5.6: Distintas configuraciones de framerate (FPS) a retransmitir a los clientes.

Configuración	1	2	3
FPS	5 fps	10 fps	25 fps

En la Figura 5.10a se puede ver que el uso de la CPU para ambos escenarios, al variar los FPS a ser retransmitidos no provocan un aumento considerable en el uso de la CPU, puesto

que tanto el procesamiento computacional de colisiones y coordinaciones, como el número de objetos en la escena sigue sin alterarse, ya que esto es lo que provoca que el uso de la CPU aumente.

En la Figura 5.10b se puede ver que el uso de la GPU se comporta parecido en ambos escenarios, puesto que necesita transmitir más *frames* por segundo, lo cual hace trabajar en gran medida a la GPU.



(a) Uso de la CPU (b) Uso de la GPU

Figura 5.10: Uso de recursos a distintos FPS a retransmitir.

En la Figura 5.11 se puede ver que el framerate se mantiene acorde con lo solicitado por los clientes, para 5 y 10 fps. Pero para 25 fps, el servidor no tiene la capacidad de retransmitir lo solicitado, y para el escenario *EPSlite* retransmite en 22 fps, y para el escenario de la ciudad en torno a 14 fps. El cuello de botella puede verse en uso de la GPU, la cual por encima del 80 % no puede suministrar más FPS al cliente.

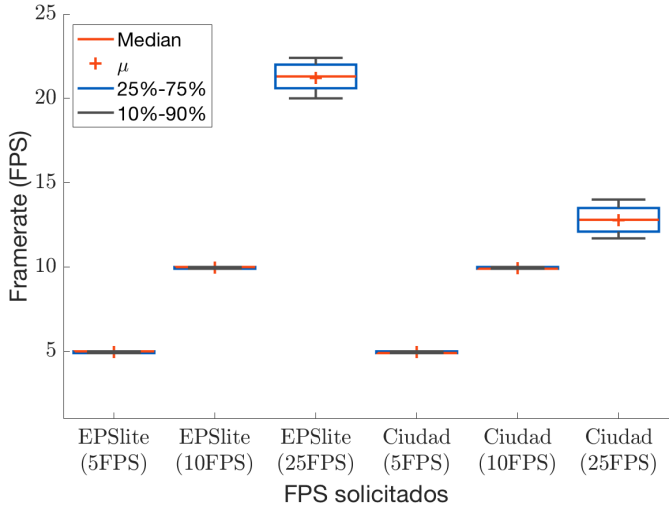


Figura 5.11: Framerate a distintos FPS a retransmitir.

5.2.5. Distinto número de cámaras

En esta evaluación se parte de una versión compilada del escenario *EPSlite*, y de otra versión compilada del escenario de la ciudad. En cada prueba de este experimento, se lanzaban tantos clientes como número de cámaras se quisiera tener retransmitiendo simultáneamente en el escenario. En la Tabla 5.6 se pueden ver las configuraciones utilizadas del experimento.

Tabla 5.7: Distinto número de cámaras retransmitiendo simultaneamente.

Configuración	1	2	3	4	5	6
Cámaras	1 cámara	2 cámaras	3 cámaras	4 cámaras	5 cámaras	6 cámaras

En la Figura 5.12 se puede ver que el uso de la CPU que para ambos escenarios, variar el número de cámaras retransmitiendo simultáneamente no provocan un aumento considerable en el uso de la CPU, puesto que tanto el procesamiento computacional de colisiones y coordinaciones, como el número de objetos en la escena sigue sin alterarse, ya que esto es lo que provoca que el uso de la CPU varíe.

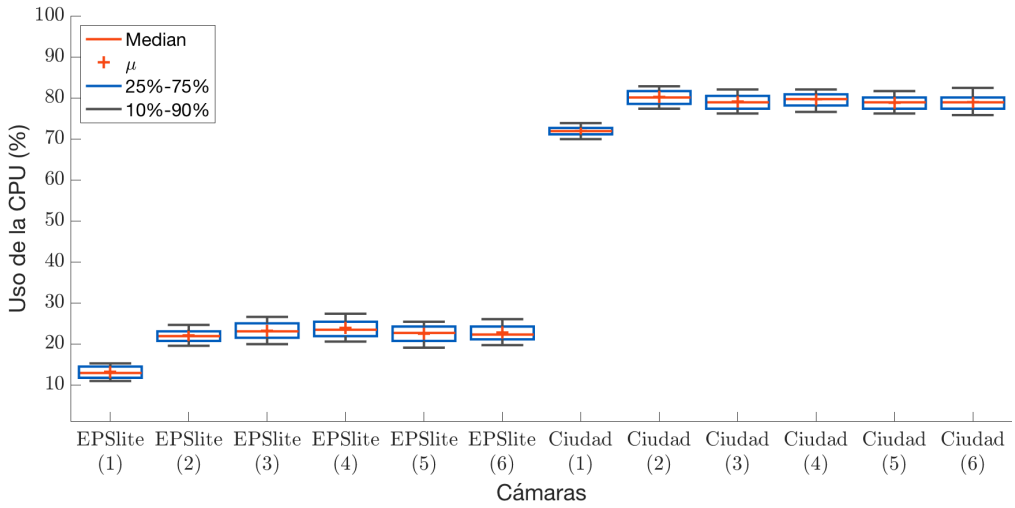


Figura 5.12: Uso de la CPU con varias cámaras retransmitiendo simultáneamente.

En la Figura 5.13 se puede ver que el uso de la GPU varía en pequeña medida para distinto número de cámaras retransmitiendo simultáneamente de los escenarios, ya que la calidad gráfica y la resolución de ellos se mantiene siempre igual (*Simple* y 640x480).

En la Figura 5.14 se puede ver que el framerate desciende conforme más cámaras añadimos a los escenarios. Este experimento guarda relación con el anterior, donde veíamos que al solicitar 25 fps para una única cámara, lo máximo que podían retransmitir al cliente eran 22 fps el escenario *EPSlite* y 14 fps el escenario Ciudad. En este caso, puede verse que el escenario *EPSlite* con dos cámaras sigue retransmitiendo sin problemas a 10 fps (22 fps entre 2 cámaras $\approx$ 11 fps), en cambio para tres cámaras retransmite a 7 fps (22 fps entre 3 cámaras $\approx$ 7 fps). De nuevo el cuello de botella puede verse en uso de la GPU, la cual por encima del 80 % no puede suministrar más FPS al cliente.

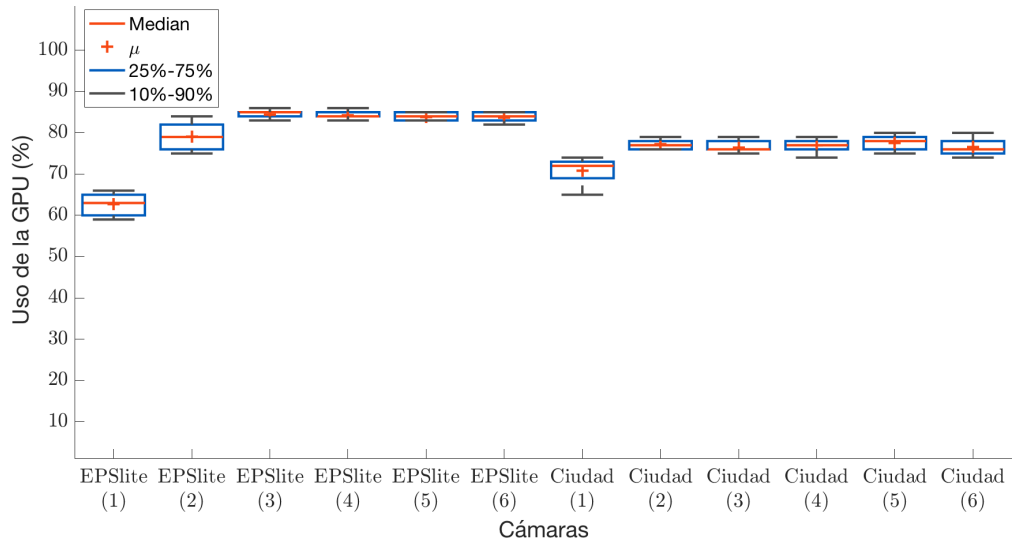


Figura 5.13: Uso de la GPU con varias cámaras retransmitiendo simultáneamente.

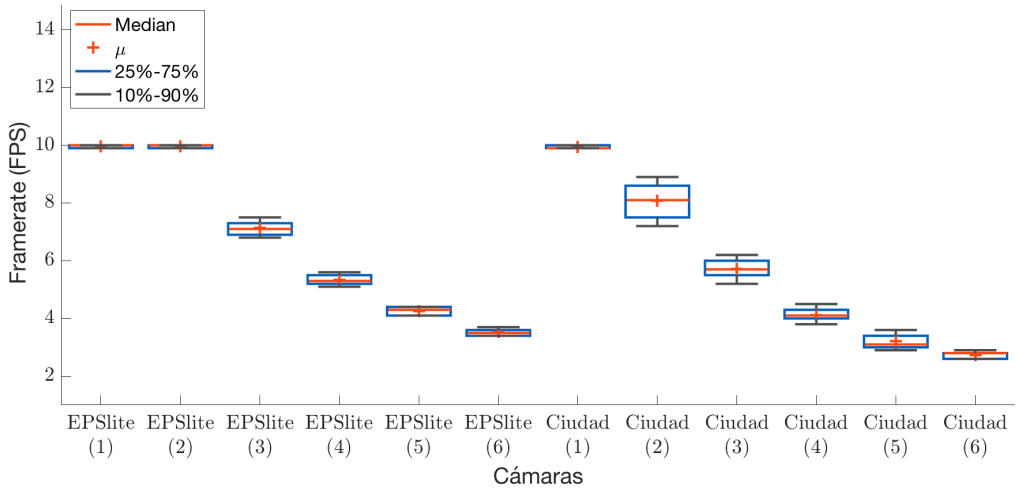


Figura 5.14: Framerate con varias cámaras retransmitiendo simultáneamente.

5.3. Conclusión

Podemos concluir que el uso de la CPU es significativamente mayor en el escenario de la ciudad que para el escenario *EPSlite*, para cualquiera de las condiciones que hemos evaluado, esto es debido a la gran cantidad de objetos dinámicos que debe controlar y gestionar el sistema que se encarga de coordinar a los vehículos y personas, además la cantidad de cálculos que debe realizar la CPU con las colisiones y deformaciones de los vehículos. En cuanto al uso de la GPU se ha llegado a la conclusión de que depende en gran medida de la calidad de los escenarios, de los FPS a ser retransmitidos y de la cantidad de luces, sombras y reflejos de los escenarios, puesto que la GPU o tarjeta gráfica, es la que se encarga del renderizado. Finalmente, el framerate se ve siempre afectado, cuando hay una gran densidad de objetos dinámicos y gran cantidad de luces, sombras y reflejos en los escenarios, cuando la calidad de los escenarios es alta (*Fantastic*) y cuando el número de cámaras retransmitiendo simultáneamente aumenta.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este documento hemos presentado un escenario que simula una ciudad donde ocurren diversas situaciones de interés, los cuales permitan diseñar, implementar y probar algoritmos para la investigación en *Computer Vision*. En el estudio de este campo existen numerosos simuladores y proyectos relacionados con el que presenta este documento: Virtual Pedestrian, MSCV, UnrealCV, Virtual KITTI, SceneNET RGB-D y PHAV no permiten colocar múltiples cámaras, además de que tanto estos simuladores, como COnGRATS no están preparados para trabajar en tiempo real. Nuestro simulador MSS y diversos simuladores utilizan el motor gráfico Unity 3D, uno de los más potentes, en cuanto a rendimiento y gráficos.

Una vez estudiado el estado del arte, se realiza una extensa búsqueda, identificación y elección de recursos disponibles, modelos 3D y algoritmos de IA de utilidad para el desarrollo de un escenario de gran calidad, a la altura de los escenarios de otros simuladores ya vistos. Se define y formaliza una taxonomía para clasificar los tipos de objetos de un escenario en función de la capacidad que tienen para controlar el inicio o la finalización de su propio movimiento. Tras definir y formalizar la generación de objetos en los escenarios (estáticos, portátiles, dinámicos y periódicos) se determinan las funcionalidades que debía tener un sistema encargado de gestionar y coordinar los objetos dinámicos (vehículos y personas) de nuestro escenario, con el fin, de reproducir situaciones y eventos de interés. Estas funcionalidades y requisitos quedan cubiertos por el sistema de tráfico (ITS), del cual se saca el máximo partido, optando por anular en un cruce la coordinación entre objetos dinámicos, con el objetivo de producir choques y atropellos entre vehículos y personas.

Una vez desarrollado el escenario, se integra el escenario desarrollado en el simulador MSS, para finalmente, evaluar su funcionalidad y rendimiento de los recursos computacionales del escenario base del simulador (*EPSlite*) frente al nuevo escenario (Ciudad), variando diversas condiciones. Llegando a la conclusión de que el uso de la CPU es mayor en el escenario Ciudad que para el escenario *EPSlite*, para cualquiera de las condiciones evaluadas. En cuanto al uso de la GPU depende considerablemente de la calidad de los escenarios, de los FPS a ser retransmitidos y de la cantidad de luces, sombras y reflejos de los escenarios. El *framerate* se ve siempre afectado, cuando hay una gran densidad de objetos dinámicos y gran cantidad de luces, sombras y reflejos en los escenarios, cuando la calidad de los escenarios es alta (*Fantastic*) y cuando el número de cámaras retransmitiendo simultáneamente aumenta.

6.2. Trabajo futuro

A la vista de los resultados que se han obtenido en este trabajo se propone trabajar en los siguientes aspectos:

- Estudiar la colocación de cámaras en un escenario, de forma, que estas no sean estáticas y fijas en un lugar determinado, si no que se puedan atribuir dichas cámaras a objetos dinámicos, tanto en vehículos (e.g. coches, interiores de autobuses o trenes, o en helicópteros), como en personas. De esta forma, se podrían conseguir grabaciones inéditas, de numerosos eventos y situaciones de interés.

- Profundizar en las acciones y las interacciones de las personas con otras personas u otros objetos dinámicos o portátiles de las escenas. Esta mejora, enriquecería considerablemente los escenarios, proporcionando mayor cantidad de situaciones de interés que en este trabajo no se han podido implementar (e.g. personas que cogen y dejan objetos, o personas que interactúan con otras).

- Compilar y ejecutar el escenario de la ciudad realizado a lo largo de este proyecto, en un equipo que cuente con una GPU de gran potencia, para posteriormente evaluar el rendimiento y los recursos que necesita en dicho equipo.

Bibliografía

[1] J. Peddie, K. Akeley, P. Debevec, E. Fonseka, M. Mangan, and M. Raphael, "A vision for computer vision: Emerging technologies," in *ACM SIGGRAPH 2016 Panels, SIGGRAPH ?16, (New York, NY, USA)*, pp. 2:1–2:2, 2016. 1

[2] E. Lantsova, T. Voitiuk, T. Zudilova, and A. Kaarna, "Using low-quality video sequences for fish detection and tracking," in *2016 SAI Computing Conference (SAI)*, pp. 426–433, Julio 2016. 1

[3] Nvidia Corporation, "The ai car computer for self-driving vehicles," 2016. 1

[4] W. Burger, Matthew J. Barth, and W. Sturzlinger, *Immersive Simulation for Computer Vision.* 5, 51

[5] H. Hattori, Vishnu N. Boddeti, K. Kitani, and T. Kanade, "Learning scene-specific pedestrian detectors without real data," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1 – 9, Junio 2015. 7, 51, 52

[6] A. Gaidon, . Q. Wang, Y. Cabon, and E. Vig, "VirtualWorlds as proxy for multi-object tracking analysis," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–10, Junio 2016. 7, 53

[7] W. Starzyk and Faisal Z. Qureshi, "Software laboratory for camera networks research," in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, pp. 284 – 293, Junio 2013. 7, 55

[8] César R. de Souza, A. Gaidon, Y. Cabon, and Antonio M. López, "Procedural generation of videos to train deep action recognition networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–20, Julio 2017. 7, 57, 58

[9] Geoffrey R. Taylor, Andrew J. Chosak, and Paul C. Brewer, "OVVV: Using virtual worlds to design and evaluate surveillance systems," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, Junio 2007. 7, 58

[10] Mario González Jiménez, "Sistema multi-cámara distribuido basado en Unity," in *Trabajo final de grado, Escuela Politécnica Superior (Universidad Autonóma de Madrid)*, Enero 2017. 8, 9, 10, 14

[11] X. Wang, "Intelligent multi-camera video surveillance: A review," in *Pattern Recognition Letters*, vol. 34, pp. 3–19, 2013. 10

[12] E. Ristani, F. Solera, Roger S. Zou, R. Cucchiara, and C. Tomasi, "Performance measures and a data set for multi-target, multi-camera tracking," in *arXiv preprint arXiv:1609.01775*, Septiembre 2016. 11

[13] Juan C. San Miguel, José M. Martínez, and A. García, "An ontology for event detection and its application in surveillance video," in *2009 Advanced Video and Signal Based Surveillance*, Septiembre 2009. 12

[14] Unity Technologies, "Unity Manual." 16

[15] I. Ouazzani, "Manual de creación de videojuego con Unity 3D," in *Proyecto Fin de Carrera, Universidad Carlos III de Madrid*, Agosto 2012. 16

[16] D. Birch, "Mapping game engines for visualisation," 2009. 45

[17] E. Araiza and S. Adavia, "Game engines overview: Natural user interaction in cars," 45

[18] D. Biedermann, M. Ochs, and R. Mester, "Evaluating visual ADAS components on the COnGRATS dataset," in *Intelligent Vehicles (IV) Symposium, Gothenburg, Sweden*, Junio 2016. 54

[19] D. Biedermann, M. Ochs, and R. Mester, "COnGRATS: Realistic simulation of traffic sequences for autonomous driving," in *Image and Vision Computing New Zealand (IVCNZ), Auckland, New Zealand*, Noviembre 2015. 54

[20] V. Veeravasarapu, C. Rothkopf, and V. Ramesh, "Model-driven simulations for computer vision," in *2017 IEEE Winter Conference on Applications of Computer Vision*, pp. 1 – 9, Marzo 2017. 54, 55

[21] Faisal Z. Qureshi and D. Terzopoulos, "Surveillance in Virtual Reality: System design and multi-camera control," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, Junio 2007. 55, 56

[22] W. Qiu and A. Yuille, "UnrealCV: Connecting computer vision to unreal engine," in *arXiv preprint arXiv:1609.01326*, 2016. 56

[23] G. Ros, L. Sellart, J. Materzynska, D. Vázquez, and Antonio M. López, "The SYNTHIA dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–10, Junio 2016. 56, 57

[24] J. McCormac, A. Handa, S. Leutenegger, and Andrew J. Davison, "SceneNet RGB-D: 5m photo-realistic images of synthetic indoor trajectories with ground truth," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 1–11, Octubre 2017. 57

[25] Luis Pérez Llorente, "Simulador virtual para sistemas multi-cámara distribuidos," in *Trabajo final de grado, Escuela Politécnica Superior (Universidad Autonóma de Madrid)*, Julio 2015. 58

Glosario

- **EPS**: Escuela Politécnica Superior
- **UAM**: Universidad Autónoma de Madrid
- **TFG**: Trabajo Fin de Grado
- **IA**: Inteligencia Artificial
- **3D**: 3 Dimensiones
- **VPU**: Video Processing and Understanding
- **MSS**: Multi-camera System Simulator
- **VAR**: Video Assistant Referee
- **SWS**: Simple Waypoint System
- **ITS**: Intelligent Traffic System
- **FPS**: Frames Per Second
- **CPU**: Central Processing Unit
- **GPU**: Graphics Processing Unit
- **RAM**: Random Access Memory
- **PTZ**: Pan Tilt Zoom
- **HD**: High Definition
- **GB**: GigaByte
- **VR**: Virtual Reality

Apéndice A

Motores gráficos

A.1. Introducción

En los últimos años, la industria de los videojuegos y de la realidad aumentada ha crecido en tal medida, que las simulaciones interactivas o físicas y las representaciones gráficas detalladas ya no necesitan ejecutarse en costosos y especializados equipos de alto procesamiento gráfico. Hoy en día, cualquiera podría ejecutarlas en su propio ordenador, videoconsola, o incluso en su propio dispositivo móvil, a un costo significativamente menor. Esto es posible debido a la uso de arquitecturas estandarizadas *(frameworks)* donde los juegos y simuladores virtuales pueden ser diseñados y creados, con mayor fácilidad y de una forma más óptima. Para propósitos de investigación, es de vital importancia un comportamiento preciso de todos los procesos involucrados, y no todos los motores gráficos nos ofrecen las mismas características, por lo que es necesario estudiar que motor gráfico es el más apropiado para una tarea específica, ya que no todos los motores gráficos ofrecen las mismas características. En la siguiente sección, estudiaremos los motores gráficos más relevantes [16, 17].

A.2. Motores gráficos más relevantes del mercado

Panda 3D[1] es un motor gráfico gratuito y de código libre compatible con codificación de scripts en C++ y Python. Tiene un propio sistema de generación de perfiles y sombreadores. Se puede desplegar en plataformas como Windows, Mac OS y Linux, pero no soporta Streaming. En la Figura A.1, se pueden ver ejemplos de juegos desarrollados con Panda 3D.

Figura A.1: Ejemplos de juegos desarrollados con Panda 3D. http://www.panda3d.org

[1] http://www.panda3d.org

Ogre 3D[2] es gratuito y de código abierto. Proporciona un SDK en C++ y un puerto para C#. Todo el trabajo con el motor requiere una gran cantidad de codificación, puesto que se codifica con el propio código sobre el que esta desarrollado. Sin embargo, admite Streaming con una codificación personalizada y es soportado en Windows, Mac OS y Linux. En la Figura A.2, se pueden ver ejemplos de juegos desarrollados con Ogre 3D.

Figura A.2: Ejemplos de juegos desarrollados con Ogre 3D. http://www.ogre3d.org

Quest 3D[3] es un motor gráfico centrado en la visualización arquitectónica. No incluye el conjunto de características más amplio del juego que otros motores gráficos que existen en el mercado, pero es compatible con una amplia gama de dispositivos de entrada y visualización, y admite publicación web. Sin embargo, carece de varias funciones de renderización de alta gama y no soporta Streaming. Se puede desplegar en Windows y en Web (Active X). En la Figura A.3, se pueden ver ejemplos de juegos desarrollados con Quest 3D.

Figura A.3: Ejemplos de juegos desarrollados con Quest 3D. http://www.quest3d.com

ShiVa 3D[4] es un motor gráfico comercial soportado en cualquier plataforma (Windows, Mac OS, Linux, IOS, Android, Windows Mobile, PlayStation, Xbox y Wii) y también cuenta con un reproductor web. Ofrece un sistema de edición, un buen conjunto de características y soporte a Streaming. En la Figura A.4, se pueden ver ejemplos de juegos desarrollados con Shiva 3D.

[2] http://www.ogre3d.org
[3] http://www.quest3d.com
[4] http://www.stonetrip.com/

Figura A.4: Ejemplo de juegos desarrollados con Shiva 3D. http://www.stonetrip.com/

Torque 3D[5] es compatible con Windows, Mac OS y reproductor Web. Proporciona numerosas características de gran calidad, incluyendo físicas PhysX y diversas funcionalidades avanzadas de renderizado e iluminación. También es compatible con un robusto sistema de publicación web. En la Figura A.5, hay ejemplos de juegos desarrollados con Torque 3D.

Figura A.5: Ejemplos de juegos desarrollados con Torque 3D. http://www.garagegames.com/products/torque-3d

Cry Engine[6] diseñado por la compañía alemana Crytek es un motor gráfico ampliamente utilizado en la industria para juegos de alta gama. Es compatible con el desarrollo en PlayStation, Xbox y PC, a su vez, contiene un generador de terreno y vegetación procedural. Es compatible con sistemas de edición e iluminación dinámica. En la Figura A.6, se pueden ver ejemplos de juegos desarrollados con Cry Engine.

Figura A.6: Ejemplos de juegos desarrollados con Cry Engine. https://www.cryengine.com

[5] http://www.garagegames.com/products/torque-3d
[6] https://www.cryengine.com

Source Engine[7] es un motor gráfico creado por la empresa Valve Corporation y desarrollado para diversas plataformas como Mac OS, Windows, Linux, Xbox y PlayStation. Debutó en junio de 2004 con el videojuego Counter-Strike: Source. Esta herramienta fue diseñada para ir evolucionando y ser escalable mientras la tecnología sigue avanzando. Cuenta con una tecnología avanzada de luces y sombras. En la Figura A.7, se pueden ver ejemplos de juegos desarrollados con Source Engine.

Figura A.7: Ejemplos de juegos desarrollados con Source Engine. https://developer.valvesoftware.com/wiki/SourceEngineFeatures

Amazon Lumberyard[8] es un motor gráfico multiplataforma gratuito basado en Cry Engine lanzado por Amazon. La principal ventaja de Amazon Lumberyard es que tiene una licencia completamente gratuita, que incluye su código fuente completo, para el desarrollo de juego para videoconsolas y PC. En contra partida, se debe usar Amazon Web Services (AWS Cloud) para juegos multijugador. La gran ventaja de este editor es que incluye una herramienta de secuencia de comandos visuales de arrastrar y soltar. En la Figura A.8, se pueden ver ejemplos de juegos desarrollados con Amazon Lumberyard.

Figura A.8: Ejemplos de juegos desarrollados con Amazon Lumberyard. https://aws.amazon.com/lumberyard/

Unreal Engine[9] es un motor de juego desarrollado por Epic Games, lanzado inicialmente en 1998. Es el motor gráfico de código abierto más utilizado en la industria del juego debido al acceso de código fuente completo que permite personalizar y extender todas las herramientas y subsistemas de Unreal. UE4 permite desplegar juegos para muchas plataformas: Windows, PlayStation, Xbox, Android, etc. Algunas características importantes son la compatibilidad

[7] https://developer.valvesoftware.com/wiki/SourceEngineFeatures
[8] https://aws.amazon.com/lumberyard/
[9] https://www.unrealengine.com/

con los gráficos API modernos (DirectX 121, Vulkan2), soporte para dispositivos de Realidad Virtual (HTC Vive3, SteamVR4), y gran calidad y expresividad para visualizaciones arquitectónicas, simulaciones o películas. En la Figura A.9, se pueden ver ejemplos de juegos desarrollados con Unreal Engine.

Figura A.9: Ejemplos de juegos desarrollados con Unreal Engine. https://www.unrealengine.com/

Unity 3D[10] es un motor gráfico multiplataforma desarrollado por Unity Technologies y se utiliza para el desarrollo de juegos para PC, videoconsolas y principalmente dispositivos móviles. Se convirtió popular al ofrecer el primer soporte multiplataforma líder en la industria, que incluye Mac Os, Windows, Linux, Android, IOS y muchas otras. Recibió la atención de la comunidad gracias a Unity Web Player: un complemento para ver contenido 3D creado con Unity directamente en su navegador web. Es la plataforma de desarrollo de VR más utilizada, admite todos los dispositivos de realidad virtual: Oculus Rift, Gear VR, Steam VR, HTC Live, Playstation VR y Microsoft HoloLens. Otras características importantes son Unity Ads, Unity Collaborate, y la compatibilidad con Blender, 3DS Max, CINEMA 4D, Maya, Softimage. En la Figura A.10, se pueden ver ejemplos de juegos desarrollados con Unity 3D.

Figura A.10: Ejemplos de juegos desarrollados con Unity 3D. https://unity3d.com/

[10] https://unity3d.com/

Apéndice B

Simuladores virtuales

B.1. Introducción

En *Computer Vision*, los simuladores virtuales proporcionan un entorno de pruebas para el diseño y desarrollo de algoritmos. Las imágenes y vídeos sintéticos a menudo se han considerado inadecuados para medir el rendimiento de los algoritmos [4], ya que, eliminaban características y detalles que las cámaras digitales eran capaz de captar y no ofrecían una suficiente simulación realista. Pero en los últimos años, con el uso de estos *frameworks*, es posible simular escenarios fotorrealistas con el fin de diseñar, desarrollar, probar y ahorrar costes y tiempos significativos. Existen diversos simuladores virtuales que cuentan con características muy diferentes, de tal forma, que pueden ser clasificados en función de los datos que utilizan para generar sus escenarios, es decir, si necesitan abastecerse de datos reales para generar los entornos simulados, o si generan y personalizan sus escenarios tal y como quieran sus desarrolladores, sin limitación alguna y sin datos reales.

B.2. Simuladores generados a partir de escenarios reales

Los simuladores a partir de datos reales obtenidos de escenarios grabados u observados en el mundo real, aportan situaciones más auténticas puesto que recrean eventos con total exactitud, tal y como ocurrieron en la realidad. El inconveniente de este tipo de simuladores es que los escenarios utilizados deben recrearse con gran similitud a la realidad, encontrándose limitaciones a la personalización.

Virtual Pedestrian

En Virtual Pedestrian[1] [5] se propone una herramienta capaz de generar virtualmente peatones en una escena de acuerdo a la geometría de la perspectiva de un escenario de interés estudiado en la realidad. Es un sistema únicamente simula peatones y sus trayectorias en escenas específicas. Una diferencia con otros simuladores es que esta herramienta utiliza datos reales como entrada para generar datos sintéticos, es decir, es necesario tener un escenario y datos reales, para después, generar un escenario virtual que guarde gran similitud con el escenario estudiado del cual se han obtenido los datos reales. Esta herramienta utiliza un

[1] http://vishnu.boddeti.net/projects/detection-by-synthesis.html

motor gráfico propio para el desarrollo y generación de escenario y peatones. En la Figura B.1, se pueden ver ejemplos de escenarios reales con sus homólogos virtuales de Virtual Pedestrian.

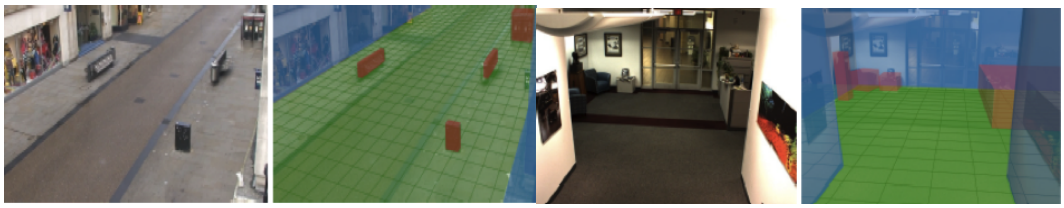


Figura B.1: Ejemplos de escenarios de Virtual Pedestrian y escenas reales. [5]

3DWM

3D World Model server (3DWM)[2] gestiona, recrea y representa a personas y objetos en el escenario 3D virtual, ya que son detectados y rastreados en el escenario real, colocando cámaras virtuales en el escenario en posiciones que se asemejen a la ubicación de las cámaras reales. De esta forma, genera imágenes y vídeos sintéticos, tal y como fueron vistos por cámaras reales, como resultado los eventos pueden ser seguidos en cámaras reales y virtuales en paralelo, la combinación de las dos vistas puede dar una visualización única y realmente eficaz. En la Figura B.2, se pueden ver ejemplos de escenarios reales con sus análogos virtuales de 3DWM.

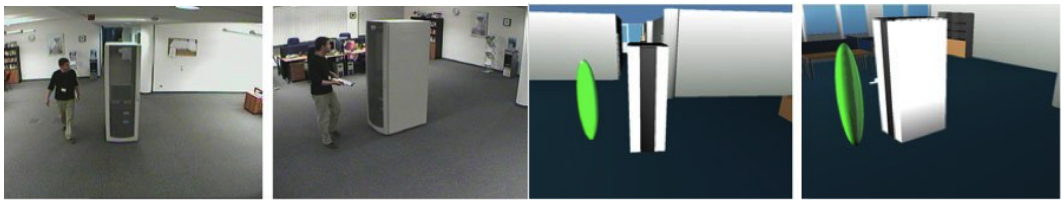


Figura B.2: Ejemplos de escenarios de 3DWM y escenas reales. https://goo.gl/4zgNYH

Mirror Worlds

Mirror Worlds[3] está enfocado para la investigación en entornos reales y virtuales. Cada entorno dentro del simulador es un sistema compuesto por un escenario real y otro virtual, donde los usuarios que actúan en el entorno real influyen en las escenas sintéticas, estando conectados los aspectos virtuales y reales entre sí mediante un sistema de redes de datos basados en coordenadas para permitir la interacción del usuario con el resto del sistema, incluidos los objetos y otros usuarios. A medida que alguien se mueve físicamente a través de un edificio en el mundo real, sus datos de localización se extraen utilizando cámaras colocadas en todo el edificio y un software de visión artificial. El número de cámaras puede configurarse, añadiéndose tantas cámaras virtuales en el escenario sintético, como cámaras existen en el mundo real, y retransmitiendo cada una de ellas de forma independiente. Cuentan con dos clientes: el primero utiliza Unity 3D como motor gráfico, y el segundo está basado en X3D. En la Figura B.3, se puede ver un ejemplo de un escenario virtual de Mirror Worlds.

[2] http://www.identrace.com/products/3d-world-model-server.html
[3] http://icat.vt.edu/MirrorWorlds/overview.html

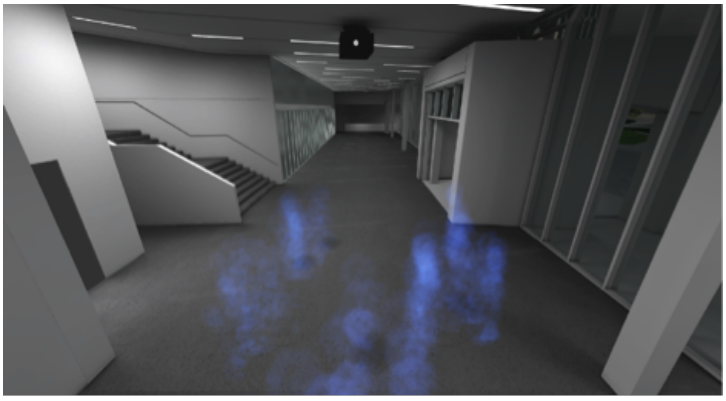


Figura B.3: Ejemplo de un escenario virtual de Mirror Worlds. https://goo.gl/L9jWLp

Virtual KITTI

Virtual KITTI[4] [6], recrea virtualmente escenarios observados en el mundo real, con el fin de aportar situaciones más auténticas, puesto que se recrean dichos eventos con total exactitud, tal y como ocurrieron en la realidad, pero la peculiaridad de este simulador, es que además, permite configurar libremente algunas condiciones y configuraciones globales del escenario, utilizando efectos especiales, sistemas de partículas, y distintas condiciones de iluminación, para simular diferentes horas del día (día, noche, amanecer, puesta del sol), y distintas condiciones meteorológicas (nublado, niebla y lluvia pesada). El gran inconveniente de este simulador es que no incluyen peatones, puesto que son más difíciles de animar, y solo consideran los automóviles como objetos de interés por su simplicidad y porque son la principal categoría de KITTI. Tienen varios mundos virtuales predefinidos desarrollados con el motor gráfico comercial de Unity 3D. Los escenarios sintéticos desarrollados, son utilizados por diversos simuladores del mercado. En la Figura B.4, se pueden ver ejemplos de escenarios reales y virtuales de Virtual KITTI.

Figura B.4: Ejemplos de escenarios de Virtual KITTI y los escenarios reales. [6]

[4] http://www.europe.naverlabs.com/Research/Computer-Vision/Proxy-Virtual-Worlds

B.3. Simuladores generados a partir de escenarios sintéticos

Los simuladores generados sintéticamente, sin estar basados en datos reales permiten a sus desarrolladores personalizar libremente los escenarios, creando y generando nuevas situaciones y pudiendo configurar libremente diversas configuraciones del escenario. En estos simuladores, hay que realizar un gran esfuerzo en que las escenas y los eventos se parezcan a la realidad, con el fin de ser de mayor utilidad en el entrenamiento de algoritmos.

COnGRATS

Computer Graphic generated syntetic Traffic Scenes (COnGRATS)[5] [18, 19] es un sistema flexible y altamente confiable, donde se pueden crear escenarios de tráfico con movimiento realista del vehículo, iluminación variable y colocación de cámaras libremente. Dicho realismo y calidad gráfica con la que cuentan sus escenarios, se debe al uso que hacen del motor gráfico de Unity 3D. El posicionamiento de la cámara es flexible y es posible utilizar tanto montajes sobre vehículos, como cámaras estáticas enfocadas hacia calles y cruces. Del mismo modo, el número de cámaras que se utilizan se puede configurar libremente, el problema es que no puede transmitir las imágenes captadas por las cámaras en tiempo real. Los escenarios cuentan con diversas condiciones ambientales, incluyendo humedad, niebla y escenarios nocturnos, para dar mas realismo a los escenarios generados. En la Figura B.5, se pueden ver ejemplos de escenarios virtuales de COnGRATS.

Figura B.5: Ejemplos de escenarios virtuales de COnGRATS. https://goo.gl/eh4Vpc

MSCV

Model-driven Simulations for Computer Vision (MSCV)[6] [20] es un simulador que emplea diversos motores de renderizado, tanto representaciones basadas en iluminación directa (e.g. sombreadores Lambertianos) y como métodos de iluminación global (e.g. *Ray-tracing* y *Monte-Carlo Path Tracing*). Pero el motor de renderizado empleado para el desarrollo de la mayoría de escenas del simulador, es el motor gráfico Blender 3D, de código abierto. Han recopilado gran cantidad modelos, y texturas de 3D CAD. Como contrapartida, no puede tener diversas cámaras retransmitiendo en tiempo real. En la Figura B.6, hay ejemplos de escenarios de MSCV.

[5] https://www.vsi.cs.uni-frankfurt.de/portfolio/congrats
[6] https://ieeexplore.ieee.org/abstract/document/7926706/

Figura B.6: Ejemplos del escenario de MSCV, con varios motores de renderizado. [20]

SLNCR

Software Laboratory for Camera Networks Research (SLCNR)[7] [7] es un simulador virtual capaz de generar vídeo sintético desde múltiples cámaras, pero sin poder retransmitir desde ellas, personalizando mundos virtuales 3D con una herramienta incluida para simular el tráfico peatonal que también apoya la detección de peatones y rastreo. De esta herramienta debe mencionarse que se puede desplegar a través de una red de equipos para simular escenas grandes y complejas. Este simulador de visión virtual tiene un motor gráfico basado en Panda3D, capaz de simular escenas en 3D incluyendo objetos dinámicos como peatones, automóviles, etc. Este simulador virtual tiene la capacidad de cambiar rápidamente la escena, el número de peatones, ubicaciones y propiedades de las cámaras, facilitando a los investigadores estudiar sus redes de cámaras en una variedad de entornos. En la Figura B.7, se pueden ver ejemplos de escenarios virtuales de SLNCR.

Figura B.7: Ejemplos del mundo virtual de SLNCR con escenarios y viandantes. [7]

SVR

Surveillance in Virtual Reality: system design and multi-camera control (SVR)[8] [21] es un sistema de simulación virtual de vigilancia que cuenta con cámaras estáticas ubicadas en una reconstrucción sintética de la estación de tren de Pennsylvania en New York City. Las cámaras son fácilmente configurables en el entorno virtual, añadiendo el número de cámaras que sean necesarias. La estación de tren está poblada por peatones virtuales animadores, pudiendo sintetizar eficientemente más de 1000 peatones realizando una rica variedad de actividades, realizando distintos eventos de interés como: entrar y salir de la estación, evitar las colisiones cuando viajan, sentarse en los bancos, comprar comida de las máquinas expendedoras, bajar o subir escaleras y entrar o salir de un tren. Cabe mencionar que el motor gráfico utilizado

[7] https://faisalqureshi.github.io/research-projects/vvs/index.html
[8] https://faisalqureshi.github.io

para la representación virtual de este simulador es OpenGL[9]. El gran inconveniente de este simulador es que ya no se encuentra disponible en el mercado. En la Figura B.8, se pueden ver ejemplos de escenarios virtuales de SVR.

Figura B.8: Ejemplos del escenario virtual de SVR desde distintos ángulos. [21]

UnrealCV

UnrealCV[10] [22] es un proyecto de código abierto para ayudar en el campo de *Computer Vision* a investigadores, con el fin de construir mundos virtuales utilizando el motor gráfico Unreal Engine 4 (UE4). Se distribuye como un plug-in que amplia la funcionalidad de UE4 con dos características: Proporciona un conjunto de comandos para interactuar con el mundo virtual, y aporta una comunicación entre UE4 y un programa externo utilizando un servidor TCP. UnrealCV no soporta múltiples cámaras y tiene otra importante desventaja, y es que no es posible generar y transmitir imágenes en tiempo real, ya que los datos se guardan en disco primero, no en la memoria principal, lo que implica altas latencias, las cuales no permiten el procesamiento en tiempo real. Pero a diferencia de otros simuladores, este genera mundos virtuales de gran calidad gráfica. En la Figura B.9 se pueden ver ejemplos de UnrealCV.

Figura B.9: Ejemplo de un escenario desarrollado con UE4 para UnrealCV. [22]

Synthia

Synthia[11] [23] se abastece de datos sintéticos de una ciudad virtual creada con la plataforma de desarrollo Unity 3D. Esta ciudad incluye los elementos más importantes presentes en ciudades, tales como: calles, carreteras, tiendas, parques y jardines, vegetación, señales de tráfico, semáforos, vehículos y peatones. Este entorno virtual permite colocar libremente cualquiera de estos elementos en la escena. La ciudad está poblada con modelos realistas de vehículos y personas. Este mundo virtual también incluye cuatro estaciones diferentes con un cambio drástico de apariencia, con nieve durante el invierno o flores durante la primavera. Además, emplea luces, sombras y reflejos dinámicos con el fin de producir diferentes

[9] https://www.opengl.org
[10] https://github.com/unrealcv/unrealcv
[11] http://synthia-dataset.net

condiciones de iluminación y simular diferentes momentos del día, incluyendo días soleados y nublado, o anochecer. Se pueden agregar múltiples cámaras y retransmitir lo que observa cada una de ellas en tiempo real. Como contrapartida, no se puede acceso al código fuente, ni tampoco se tiene la posibilidad de personalizar los escenarios o modificar sus configuraciones. Los escenarios sintéticos desarrollados, son utilizados por diversos simuladores del mercado. En la Figura B.10, hay ejemplos de escenarios de Synthia.

Figura B.10: Ejemplos de escenarios virtuales en cada estación de SYNTHIA. [23]

SceneNET RGB-D

SceneNet RGB-D[12] [24] utiliza diseños de escenas de código abierto. Para escenarios de exterior, utilizan el conjuntos de datos de SYNTHIA, en cambio, para escenarios de interior, utilizan escenas desarrolladas por el simulador UnrealCV. Las escenas de esta herramienta son estáticas, lo que presenta una gran limitación, puesto que las escenas dinámicas proporcionarían una representación más fiel del mundo real. Aun que este simulador emplee conjuntos de datos de SYNTHIA para los escenarios de exterior, evitan los objetos dinámicos, y no utilizan la lógica de cámaras simultáneas que proporciona. En la Figura B.11, se pueden ver ejemplos de escenarios virtuales de SceneNet RGB-D.

Figura B.11: Ejemplos de un escenario de interior de SceneNET RGB-D. [24]

PHAV

Procedural Human Action Videos (PHAV)[13] [8] es un simulador que utiliza tanto escenarios completos, texturas y modelos 3D de objetos prefabricados obtenidos del Asset Store de Unity, como distintas escenas del conjunto de datos de Virtual KITTI, o de SYNTHIA. Pueden descargarse distintos conjuntos de datos y escenarios, así como su código fuente de forma gratuita desde su web. A su vez, permite uno o más actores, así como uno o más personajes secundarios, los cuales tienen numerosas acciones. Su mundo virtual se compone de grandes áreas urbanas, entornos naturales (bosques, lagos y parques), escenarios de interior y campos deportivos. Cada uno de estos escenarios puede contener viandantes y objetos de

[12] https://arxiv.org/abs/1612.05079
[13] http://adas.cvc.uab.es/phav/

fondo móviles o estáticos con los que los peatones puedan interactuar físicamente. El clima y el periodo del día son características del escenario configurables: lluvioso, nublado, claro o niebla, día o noche. En la Figura B.12, se pueden ver ejemplos de escenarios virtuales.

Figura B.12: Ejemplos de los distintos escenarios y eventos de interés de PHAV. [8]

OVVV

ObjectVideo Virtual Video (OVVV)[14] [9] es una herramienta del simulador para generar el vídeo virtual con múltiples cámaras en un mundo 3D virtual. El motor gráfico de este simulador está basado en Source Engine. OVVV está distribuido gratuitamente para fines de investigación y desarrollo. Sin embargo, ya que uno de sus componentes se basa en un juego comercial, es necesario tener una licencia. Comprando Half Life 2, se puede obtener el motor de Source SDK (incluyendo el editor de mapas y el código fuente) de forma gratuita. Sus características principales son: Permite colocar libremente y configurar parámetros independientemente para cada cámara del escenario virtual, aporta flujos de vídeo sincronizados en tiempo real de múltiples cámaras independientes observando el mismo mundo virtual, y cuenta con generación automática del terreno y pruebas repetibles y reproducibles. En la Figura B.13, se pueden ver ejemplos de escenarios virtuales de OVVV.

Figura B.13: Ejemplos de distintos escenarios y eventos de interés de OVVV. [9]

OVVV extendido

El principal problema del simulador OVVV es la ausencia de manejo de cámaras de forma distribuida. Para construir un simulador completo, en 2015, Luis Pérez presentó una extensión del simulador de OVVV[15] [25] para su tesis de licenciatura. Esta extensión incorpora a OVVV una arquitectura Cliente-Servidor que permite la comunicación desde el simulador a aplicaciones externas. Además, se desarrolló un controlador de cámaras que tiene métodos para crear, eliminar y configurar múltiples cámaras en tiempo real y simultáneamente. También proporciona mediante la biblioteca OpenCV[16] una función para visualizar.

[14] http://objectvideo-virtual-video-sdk.software.informer.com
[15] https://repositorio.uam.es/handle/10486/668674
[16] http://opencv.org/

Apéndice C

Estudio de recursos disponibles

A continuación se muestra el estudio completo de los recursos disponibles que han sido de gran utilidad en este proyecto, y que pueden ser de utilidad en futuros proyectos, en los cuales se quieran realizar nuevos escenarios con distintas ambientaciones, o se quiera incluir nueva funcionalidad.

Realizando el estudio de recursos disponibles en el Asset Store de Unity y en diversas páginas de internet, descubrimos un producto que venden Humanity 3D[1] que es de alta calidad y que puede resultar de gran utilidad en proyectos como este, los cuales se centren en las interacciones y las acciones que realizan las personas. Existen páginas adicionales que pueden ser de utilidad para la descarga de modelos 3D[2][3][4][5]

[1] http://www.humanity3d.ca/product/urban-life/
[2] https://free3d.com
[3] https://www.cgtrader.com/3d-models
[4] https://renderpeople.com/free-3d-model/
[5] http://www.turbosquid.com/Search/3D-Models/free

C.1. Escenarios completos

Tabla C.1: Tabla que muestra los escenarios completos (I).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Set Builder: New York	Sinister Games (Asset Store - Unity)	$25	Buena	https://goo.gl/sx9QcD		Cruce de calles	Escenario completo de una ciudad
Modern City Pack	Noirfx 3D Artist (Asset Store - Unity)	$65	Alta	https://goo.gl/trenoj		Cruce de calles	Escenario con números detalles (incluye día y noche)
City Night Construction Set	Zealous Interactive (Asset Store - Unity)	$40	Buena	https://goo.gl/mnJvME		Cruce de calles	Escenario completo de una ciudad por la noche
City Environment Pack	Bay Chezer (Asset Store - Unity)	$25	Buena	https://goo.gl/42NYSz		Cruce de calles	Escenario completo de una ciudad
City Pack - Modular and Tileable	3Lane Studios (Asset Store - Unity)	$35	Buena	https://goo.gl/wEjZtG		Cruce de calles	Escenario completo de una ciudad con grandes edificios
City Highway	VIS Games (Asset Store - Unity)	$25	Media	https://goo.gl/2Q2ZEv		Cruce de calles	Escenario que contiene unas carreteras y pocos edificios

Tabla C.2: Tabla que muestra los escenarios completos (II).

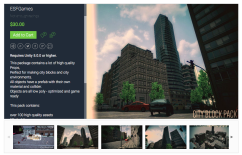
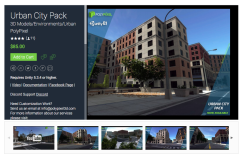
Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
City Block Pack	ESFGames (Asset Store - Unity)	$25	Media	https://goo.gl/uPA7YY		Cruce de calles	Escenario completo de una ciudad
City Night	Shadow Art (Asset Store - Unity)	$10	Buena	https://goo.gl/wYziCU		Cruce de calles	Escenario completo de una ciudad por la noche
Industrial City	Poly Pixel (Asset Store - Unity)	$80	Alta	https://goo.gl/4wWiXe		Cruce de calles	Escenario de una ciudad industrial con numerosos detalles
Urban City Pack	Poly Pixel (Asset Store - Unity)	$85	Alta	https://goo.gl/2ZbqTr		Cruce de calles	Escenario de una ciudad con numerosos detalles
Big City	To Bots (Asset Store - Unity)	$35	Buena	https://goo.gl/G3w5We		Cruce de calles	Escenario completo de una ciudad
PQ Modern City	Phasequad (Asset Store - Unity)	$75	Buena	https://goo.gl/oNMVDc		Cruce de calles	Escenario completo de una ciudad con grandes edificios

Tabla C.3: Tabla que muestra los escenarios completos (III).

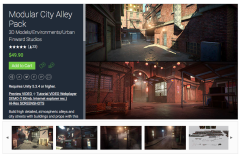
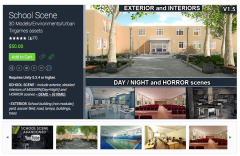
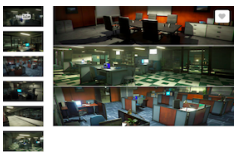
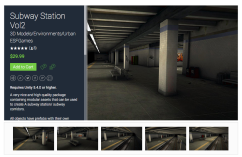
Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Modular City Alley Pack	Finward Studios (Asset Store - Unity)	$50	Alta	https://goo.gl/ED1yZH		Cruce de calles	Escenario de una ciudad con gran detalle de luces y sombras
Suburb Neighborhood House Pack (Modular)	Finward Studios (Asset Store - Unity)	$89	Alta	https://goo.gl/kL13ct		Cruce de calles	Escenario de una ciudad residencial con gran detalle
City Park Exterior Props	It's Chill Bro Studios (Asset Store - Unity)	$25	Alta	https://goo.gl/mN1SEr		Lugares públicos (Parque)	Escenario que simula un parque con gran detalle
School Scene	Tirgames assets (Asset Store - Unity)	$50	Alta	https://goo.gl/GcnCGz		Lugares públicos (Escuela)	Escenario que simula una escuela con gran detalle
Office and Police Station Pack (Modular)	Finward Studios (Asset Store - Unity)	$79	Alta	https://goo.gl/gvzhBs		Lugares públicos (Oficina de policía)	Escenario de alta calidad que simula una comisaría de policía
Titanic	AGLOBEX (Asset Store - Unity)	$199	Alta	https://goo.gl/xWTzjy		Lugares públicos (Barco)	Es un escenario de alta calidad y detalles simulando el Titanic

Tabla C.4: Tabla que muestra los escenarios completos (IV).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Subway Environment	EFSGames (Asset Store - Unity)	$25	Buena	https://goo.gl/9HWjd2		Estación de tren	Incluye vias, andenes y zonas de espera de una estación de trenes
Subway Station Vol2	EFSGames (Asset Store - Unity)	$30	Buena	https://goo.gl/p768HW		Estación de tren	Incluye vias, andenes y zonas de espera de una estación de trenes
Urban Underground	Gabro Media (Asset Store - Unity)	$45	Alta	https://goo.gl/RbsYFc		Estación de tren	Incluye vias, andenes y zonas de espera de una estación de trenes con números detalles (trenes en movimiento)
Real Metro	AGLOBEX (Asset Store - Unity)	$149	Alta	https://goo.gl/NGUDA9		Estación de tren	Escenario de altísima calidad con luces y sombras muy detalladas y trenes en movimiento

C.2. Objetos estáticos

Tabla C.5: Tabla que muestra los objetos estáticos (I).

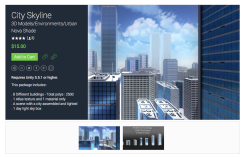
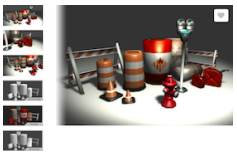
Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Simple Modular Street Kit	Jacek Jancowski (Asset Store - Unity)	$0	Media	https://goo.gl/i3Ro6Z		Cruce de calles	Solo incluye calles y aceras
Buildings Mini Pack	BSP (Asset Store - Unity)	$5	Media	https://goo.gl/hPWqrS		Cruce de calles	Se incluyen algunos edificios
Town Houses Pack	Chermandir Kun (Asset Store - Unity)	$0	Buena	https://goo.gl/G5UuBE		Cruce de calles	Se incluyen algunos edificios
City Low Poly	dactilardesign (Asset Store - Unity)	$30	Media	https://goo.gl/om2eD2		Cruce de calles	Se incluyen algunos edificios de una ciudad
City Skyline	Nova Shade (Asset Store - Unity)	$15	Buena	https://goo.gl/L76xRr		Cruce de calles	Se incluyen algunos edificios y rascacielos de una ciudad
City Houses	VIS Games (Asset Store - Unity)	$10	Media	https://goo.gl/7RLuU8		Cruce de calles	Se incluyen algunas casa y edificios residenciales bajos de una ciudad

Tabla C.6: Tabla que muestra los objetos estáticos (II).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
City Street Props Package	Alan lucker (Asset Store - Unity)	$4	Media	https://goo.gl/D9PPwT		Cruce de calles	Se incluyen objetos básicos para dar más realismo a los escenarios de ciudades
Modern City Model Pack	DEXSOFT - Games (Asset Store - Unity)	$30	Media	https://goo.gl/3ftQaa		Cruce de calles	Se incluyen objetos básicos y edificios para escenarios de ciudades
Detailed City Pack	dactilardesign (Asset Store - Unity)	$85	Media	https://goo.gl/ZnRdwB		Cruce de calles	Se incluyen objetos básicos y edificios para escenarios de ciudades
Street Assets	Remi Storms (Asset Store - Unity)	$0	Media	https://goo.gl/p6ENkv		Cruce de calles	Se incluyen objetos básicos para dar más realismo a los escenarios de ciudades

C.3. Objetos dinámicos

Tabla C.7: Tabla que muestra los objetos dinámicos (personas) (I).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Modern People	Davidb informatique (Asset Store - Unity)	$10	Media	https://goo.gl/WRgwbL		Cualquier escenario	Personas con calidad aceptable y movimientos suficientes
Modern People 2	Davidb informatique (Asset Store - Unity)	$10	Media	https://goo.gl/tWgGYY		Cualquier escenario	Personas con calidad aceptable y movimientos suficientes
Real People: Males	3DRT.com (Asset Store - Unity)	$99	Media	https://goo.gl/Ma2Uqo		Cualquier escenario	Personas (hombres) con calidad aceptable y numerosos movimientos Muy caro.
Real People: Females	3DRT.com (Asset Store - Unity)	$99	Media	https://goo.gl/2S6FXW		Cualquier escenario	Personas (mujeres) con calidad aceptable y numerosos movimientos Muy caro.
City Urban Character Pack V1	ElkanSoft (Asset Store - Unity)	$12	Media	https://goo.gl/zcnKnG		Cualquier escenario	Personas con calidad aceptable y numerosos movimientos
Urban Low Poly People	AGLOBEX (Asset Store - Unity)	$55	Media	https://goo.gl/7pLfZk		Cualquier escenario	Personas con calidad aceptable

Tabla C.8: Tabla que muestra los objetos dinámicos (personas) (II).

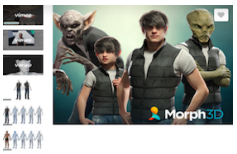
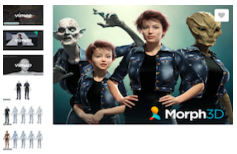
Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
UMA 2 - Unity Multipurpose Avatar	UMA Steering Group (Asset Store - Unity)	$0	Alta	https://goo.gl/cNdAZY		Cualquier escenario	Personas (hombres y mujeres) con alta calidad, las cuales pueden ser editadas
Renderpeople Free Rigged Models	Renderpeople (Asset Store - Unity)	$0	Alta	https://goo.gl/FfzWkX		Cualquier escenario	Incluye dos personas (hombre y mujer) de alta calidad
Man in a Suit	Studio New Punch (Asset Store - Unity)	$0	Buena	https://goo.gl/b5pN8y		Cualquier escenario	Incluye una persona (hombre) de buena calidad
Sci Fi Officer Captain	All * Star Characters (Asset Store - Unity)	$0	Media	https://goo.gl/bqmV92		Cualquier escenario	Incluye una persona (hombre) editable con distintas texturas y armas
MCS Male	MORPH 3D (Asset Store - Unity)	$0	Alta	https://goo.gl/G8XDC6		Cualquier escenario	Personas (hombres) con alta calidad, las cuales pueden ser editadas
MCS Female	MORPH 3D (Asset Store - Unity)	$0	Alta	https://goo.gl/mx6i23		Cualquier escenario	Personas (mujeres) con alta calidad, las cuales pueden ser editadas

Tabla C.9: Tabla que muestra los objetos dinámicos (vehículos) (I).

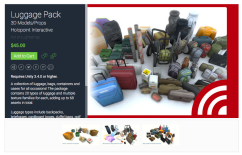
Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Vehicle Car	MPGames (Asset Store - Unity)	$2	Buena	https://goo.gl/Y42iv3		Cruce de calles	Simula un Audi
Van Vehicle	Samuel Abyan (Asset Store - Unity)	$9	Alta	https://goo.gl/DthBqh		Cruce de calles	Simula una furgoneta
Police Car & Helicopter	SICS Games (Asset Store - Unity)	$0	Buena	https://goo.gl/g5WRex		Cruce de calles	Simula un coche de policía (también incluye un helicoptero)
Cars Free – Muscle Car Pack	Super Icon Ltd (Asset Store - Unity)	$0	Buena	https://goo.gl/DX4BZF		Cruce de calles	Simula un coche americano
Sport Car – 3D model	Andrej Mikus (Asset Store - Unity)	$0	Alta	https://goo.gl/xeZX3v		Cruce de calles	Simula un Lamborghini
GR3D Sport Car 042016SSCR	Game Ready 3D Models (Asset Store - Unity)	$10	Alta	https://goo.gl/LLZQTW		Cruce de calles	Simula un Mini Cooper (de este mismo editor pueden obtenerse gran cantidad de vehículos)

Tabla C.10: Tabla que muestra los objetos dinámicos (vehículos) (II).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
3D Low Poly Car For Games	Ruslan (Asset Store - Unity)	$0	Buena	https://goo.gl/ZHP6cu		Cruce de calles	Simula un Chrysler
Single Detailed Truck	VIS Games (Asset Store - Unity)	$0	Buena	https://goo.gl/UMcCzW		Cruce de calles	Simula un camión
New York Taxi Car	Samuel Abyan (Asset Store - Unity)	$9	Alta	https://goo.gl/ppgUPM		Cruce de calles	Simula un taxi
City Bus	Rescue 3D (Asset Store - Unity)	$10	Alta	https://goo.gl/LcHPQb		Cruce de calles	Simula un autobus urbano
Mobile HQ Vehicles - Vol.1	AGLOBEX-Mobile (Asset Store - Unity)	$385	Alta	https://goo.gl/DjwVU9		Cruce de calles	Vehículos de todo tipo con gran calidad y cantidad de detalles
London Tube	dafun (Asset Store - Unity)	$59	Alta	https://goo.gl/gT7PVN		Estación de tren	Simula el metro de Londres

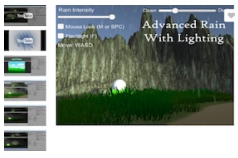
C.4. Objetos portátiles

Tabla C.11: Tabla que muestra los objetos portátiles.

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Luggage pack	VIS Games (Asset Store - Unity)	$2	Media	https://goo.gl/pja9S2		Cualquier escenario	Maletas y bolsos útiles para algunos eventos de interés
Luggage pack	Holopoint Interactive (Asset Store - Unity)	$45	Alta	https://goo.gl/xDCw9S		Cualquier escenario	Gran conjunto de maletas, bolsos y mochilas útiles para algunos eventos de interés

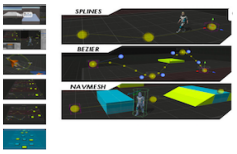
C.5. Objetos periódicos

Tabla C.12: Tabla que muestra los objetos periódicos (Incorpóreos).

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Rain Maker - 2D and 3D Rain Particle System for Unity	Digital Ruby (Jeff Johnson) (Asset Store - Unity)	$0	Media	https://goo.gl/9kg3gr		Escenarios de exterior	Simula una lluvia que se puede variar la densidad de precipitación

C.6. Sistemas y herramientas

Tabla C.13: Tabla que muestra las sistemas y herramientas.

Recurso	Editor	Precio	Calidad	Enlace	Ejemplo	Escenario	Comentarios
Truss Physics	HeartBroken (Asset Store - Unity)	$100	Alta	https://goo.gl/db5xyy		Cualquier escenario	Físicas que permiten deformaciones realistas (Versión de prueba)
Edy's Vehicle Physics	Edy (Asset Store - Unity)	$60	Media	https://goo.gl/Q6gTr3		Cruce de calles	Físicas que permiten deformaciones realistas en vehículos
Urban Traffic System PRO	AGLOBEX (Asset Store - Unity)	$499	Alta	https://goo.gl/7dYk4B		Cruce de calles	Sistema de tráfico de vehículos y personas de alta calidad
iTS - Intelligent Traffic System (Source)	Jose Garrido (Asset Store - Unity)	$200	Alta	https://goo.gl/nzgcsD		Cruce de calles	Sistema de tráfico de vehículos y personas de alta calidad e incluye lógica de semáforos
Population System PRO	AGLOBEX (Asset Store - Unity)	$149	Alta	https://goo.gl/RZsHjC		Cualquier escenario	Sistema de tráfico de personas de alta calidad
Simple Waypoint System	Rebound Games (Asset Store - Unity)	$15	Media	https://goo.gl/u8usfq		Cualquier escenario	Trayectorias para objetos dinámicos

Apéndice D

Tutoriales

En este apéndice se pretende explicar como se han implementado los distintos objetos que conforman un escenario, y los elementos mínimos que debe tener cada uno de ellos, explicados en la sección 4. Las explicaciones se han detallado paso a paso, incluyendo capturas de pantalla para mejorar la comprensión de los tutoriales, además, se incluyen referencias a los vídeos, los cuales pueden aportar mayor ayuda para llevar a cabo la realización de los mismos. También se explica la importancia del uso de "prefabs" a la hora de desarrollar escenarios basados en Unity 3D.

D.1. Creación y manejo de Prefabs

Una manera sencilla y rápida que nos proporciona Unity para almacenar un objeto GameObject con todos sus componentes, elementos, objetos internos y propiedades, son unos tipos de *Asset* llamado *Prefab*. El *Prefab* actúa como una plantilla de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un *Prefab* será inmediatamente reflejado en todas las instancias producidas por él, pero, también se puede anular componentes y ajustes para cada instancia individualmente.

- Creación de un *Prefab*: Debemos abrir el desplegable de la pestaña del menú de herramientas de Unity (*Assets* > *Create* > *Prefab*), seguidamente lo nombraremos como deseemos y finalmente arrastraremos el objeto *GameObject* que deseemos almacenar desde la ventana *Hierarchy* al *Prefab* que está vacío que se puede visualizar en la ventana *Project.*

- Instancia de un *Prefab*: Arrastrando el *Prefab* previamente creado o importado de la ventana *Project* a la ventana *Hierarchy* o *Scene*, se creará un objeto con las mismas características que el *Prefab.*

- Edición de un *Prefab*: Para editar un *Prefab* y producir los cambios en todos los objetos instanciados en el escenario con dicho *Prefab*, tan solo deberíamos modificar los componentes, elementos, objetos internos y propiedades del propio *Prefab* de la ventana *Project.* Si solo se quisiera editar un objeto instanciado con dicho *Prefab* sin que se produzcan cambios en el resto de objeto instanciados con el mismo *Prefab*, deberíamos editar el propio objeto de la ventana *Hierarchy* tal y como modificaríamos dichos

componentes de cualquier otro objeto.

Para simplificar el desarrollo de futuros proyectos que guarden relación directa con este, objetos estáticos, portátiles, periódicos y dinámicos, de ambos escenarios (cruce de calles y estación de trenes) han sido almacenados en *Prefabs*.

D.2. Tutorial de creación de un objeto estático

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto estático **(Calle)** con un *Mesh* preexistente. Para más información, puede visitar el link de YouTube[1], donde podrá ver un videotutorial realizando paso por paso, la creación de un objeto estático.

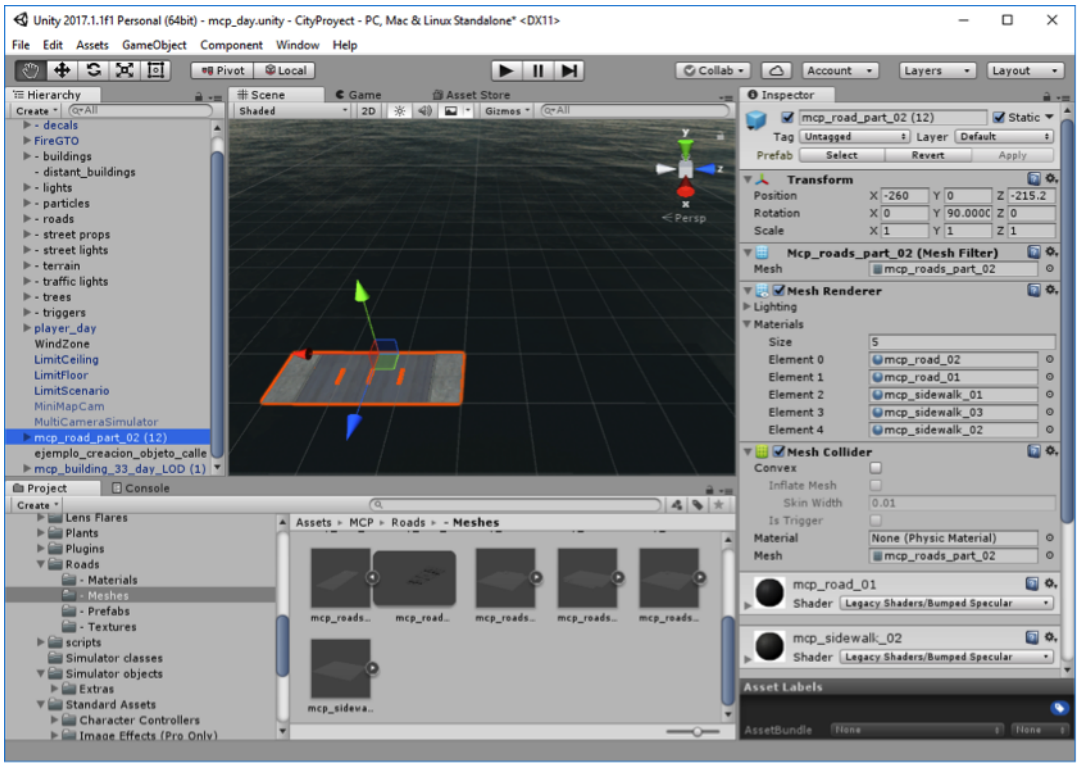


Figura D.1: Paso 1 del tutorial de creación de objeto estático.

Para crear desde cero el objeto estático **Calle** debemos añadir un *GameObject > Create Empty* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto calle*). Las propiedades de dicho objeto se pueden ver en la pestaña *Inspector*, donde se puede ver como tiene predefinido el componente *Transform* donde se puede modificar su posición, rotación y escala.

[1] https://www.youtube.com/watch?v=EYsJFjA0aE

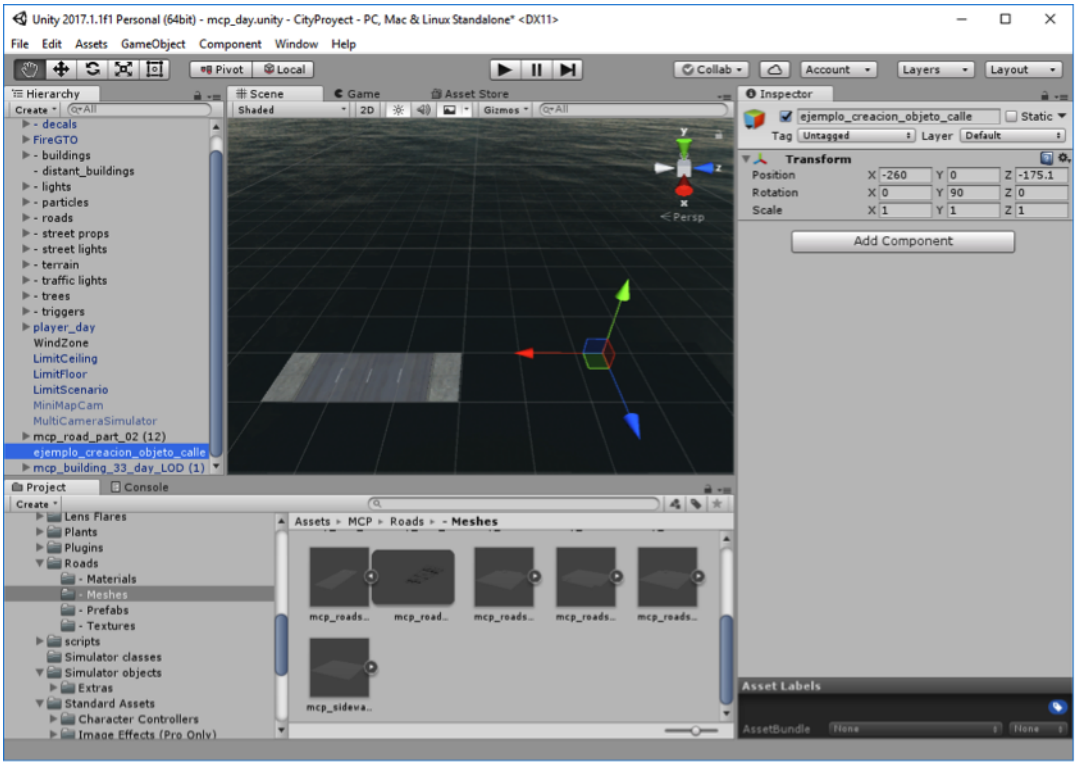


Figura D.2: Paso 2 del tutorial de creación de objeto estático.

Antes de añadir los componentes necesarios para la elaboración de un objeto estático, se debería de crear el *Mesh*, también llamado modelo geométrico o malla del objeto, mediante aplicaciones de modelado 3D como Blender o 3D Studio Max, ya que Unity no ofrece herramientas de modelización. En cambio, nos permite importar dichos modelos geométricos, soportando lectura de distintos tipos de archivo, como .fbx, .obj .dae, .3DS, .dxf, .Max (de 3D Studio Max) y .Blend (de Blender). En la página oficial se puede obtener la aplicación de modelado 3D Blender[2], y para mayor información, se puede ver su documentación[3].

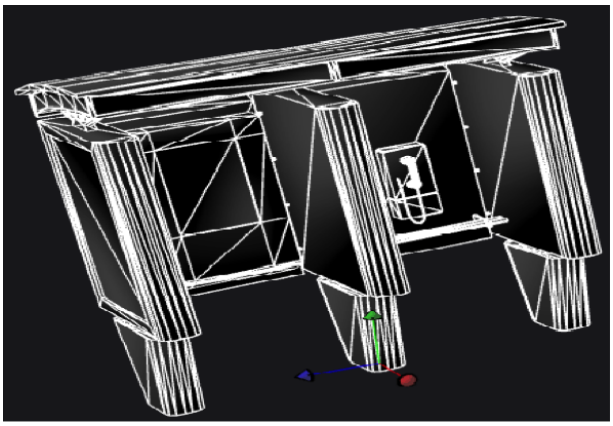


Figura D.3: Ejemplo de modelo geométrico o "Mesh" de una cabina telefónica.

Si no se desea crear un modelo geométrico o malla (también llamado *Mesh* o *Mesh Asset*),

[2] https://www.blender.org/download/
[3] https://docs.blender.org/manual/

se pueden obtener del Asset Store de Unity, o de cualquier otra página de descarga de modelos 3D, con cualquiera de los archivos que permite Unity importar explicados anteriormente. En este ejemplo, partiremos de los modelos geométricos que importamos junto con el escenario del cruce de calles descargado del Asset Store de Unity[4]. Si se deseasen editar o modificar los modelos geométricos descargadas, se debería de hacer uso de alguna de las aplicaciones o herramientas de modelado 3D con las que sea compatible el *Mesh* descargado del Asset Store de Unity.

A continuación, se deben añadir al objeto tres componentes más, los cuales cubrirán los elementos que necesita todo objeto estático:

- **Mesh Filter:** Este componente toma un modelo geométrico (*Mesh Asset*) y se la pasa al componente *Mesh Renderer* para renderizar y visualizar el objeto, es decir, este componente lo único que hace es agregar e importar al objeto el modelo geométrico deseado, moldeándolo tal y como indica la forma que se le ha añadido. De la siguiente forma se podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Filter.*

- **Mesh Renderer:** Este componente toma el modelo geométrico del componente *Mesh Filter* y la renderiza el objeto en la posición definida por el componente *Transform* del objeto. El componente *Mesh Renderer* guarda gran relación con los materiales, puesto que, a partir de ellos, este componente define como es mostrado y visualizado dicho objeto. De la siguiente forma se podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Renderer.*

- **Mesh Collider (Box Collider, Sphere Collider o Capsule Collider):** Este componente toma un modelo geométrico (*Mesh Asset*) y construye su área o volumen de colisiones (*Collider*) basándose en ese *Mesh*. Por lo tanto, el componente *Mesh Collider* construye la representación de colisión del *Mesh* adjunto al *GameObject*, y extrae las propiedades del componente *Transform* para configurar su posición y escala correctamente. Una gran ventaja de este componente es que la forma del *Collider* puede ser exactamente la misma como la forma del *Mesh* visible del objeto, resultando colisiones más auténticas y precisas. Sin embargo, esta precisión conlleva una mayor sobrecarga de procesamiento que las colisiones, por lo que se puede optar por un modelo geométrico similar, más sencillo, que aporte un rendimiento adecuado, como *Box Collider, Sphere Collider o Capsule Collider*, en los que solo haría falta añadir las medidas para ajustarlo correctamente al objeto. De la siguiente forma se podrá añadir este componente el objeto: *Add Component > Physics > Mesh Collider (Box Collider, Sphere Collider o Capsule Collider).*

[4] https://assetstore.unity.com/packages/3d/environments/urban/modern-city-pack-18005

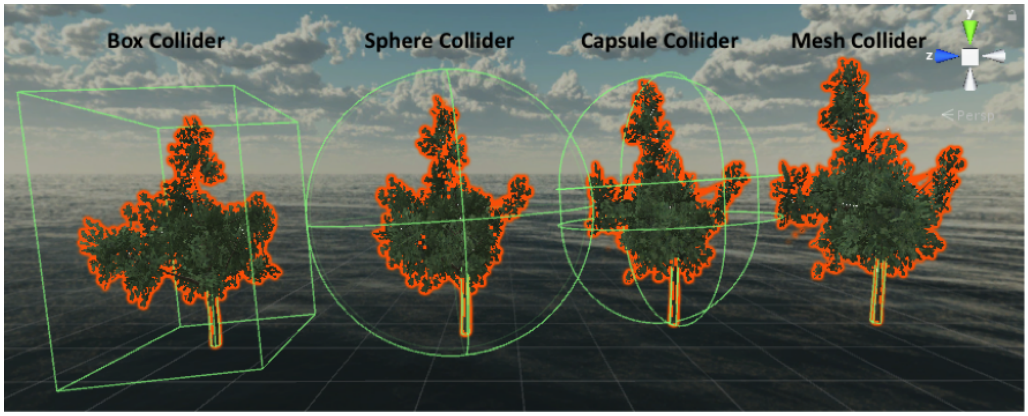


Figura D.4: Ejemplo comparativo entre distintas áreas de colisiones de un árbol.

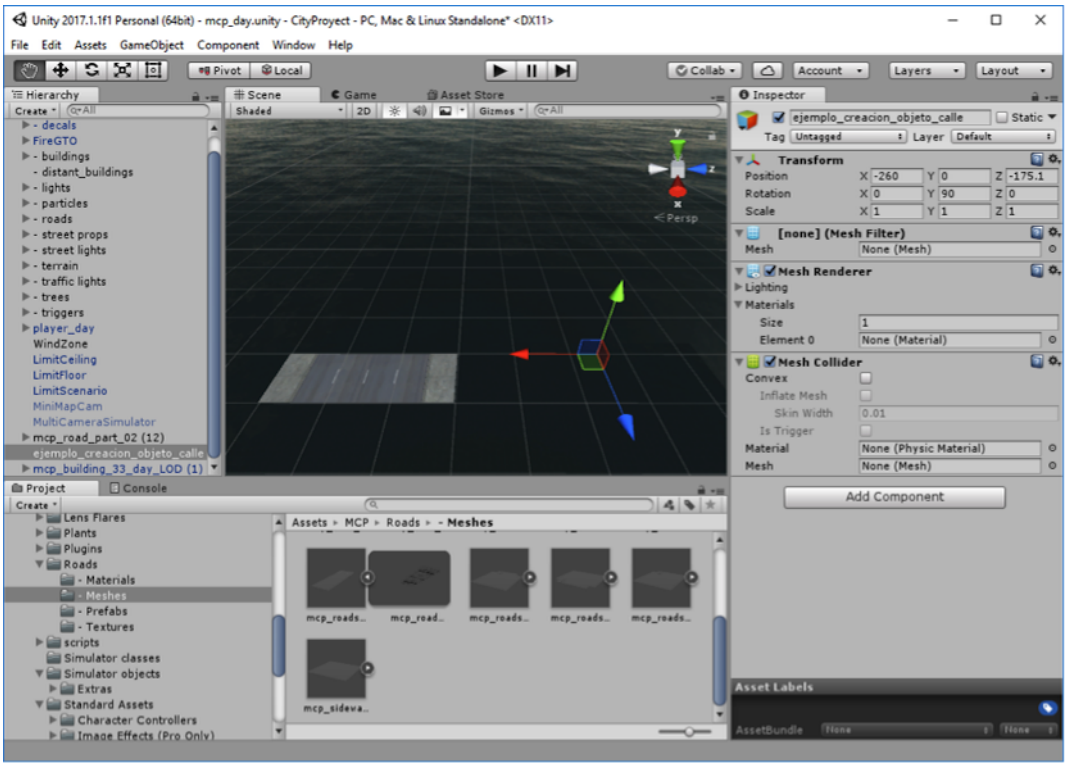


Figura D.5: Paso 3 del tutorial de creación de objeto estático.

Añadimos tanto en el componente *Mesh Filter*, como en el componente *Mesh Collider* el mismo modelo geométrico ya importado en formato .fbx de ese tipo de calle (*mcp roads part 02*). En esta parte es en la que se puede añadir cualquier otro modelo geométrico realizado mediante una herramienta de modelado 3D externa a Unity, y añadir dicho *Mesh*, en lugar del modelo geométrico indicado.

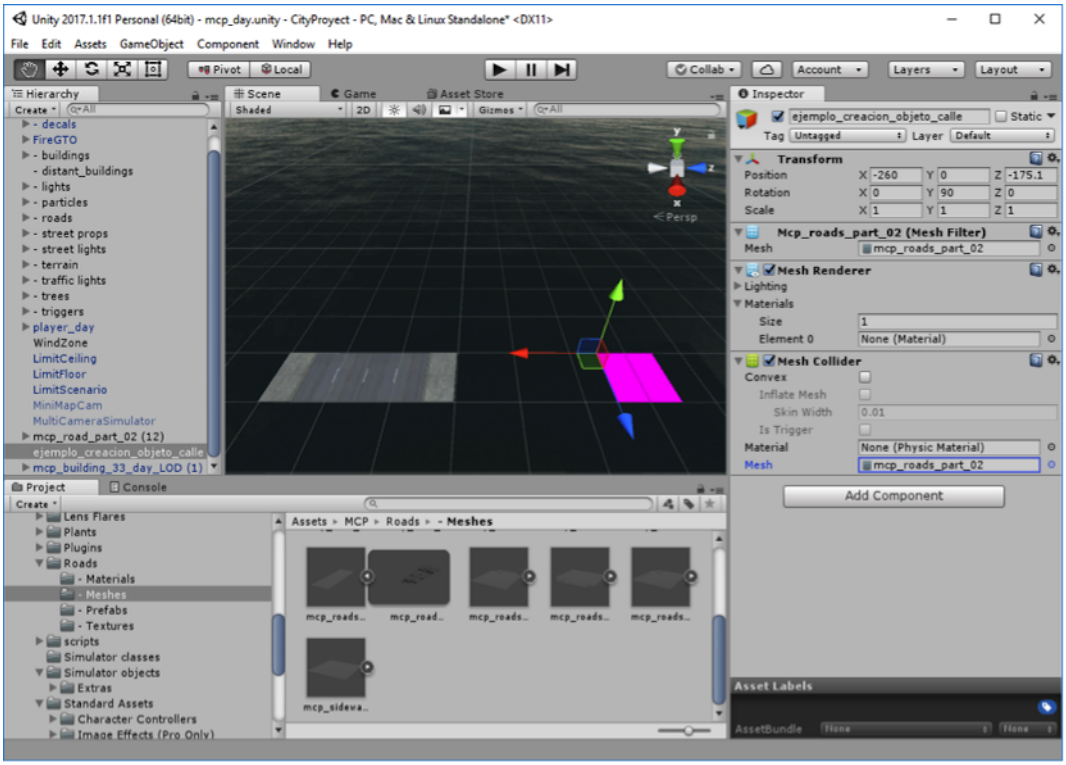


Figura D.6: Paso 4 del tutorial de creación de objeto estático.

Para este modelo geométrico que hemos añadido al objeto se requieren cinco materiales,es decir, que el número de materiales está relacionado con el modelo geométrico del objeto, y por lo tanto debemos aumentar el tamaño del conjunto de materiales a cinco ($Size = 5$), y añadir cada material que deseemos a cada parte del modelo. En este caso le añadimos los materiales de la calle objetivo que queremos obtener.

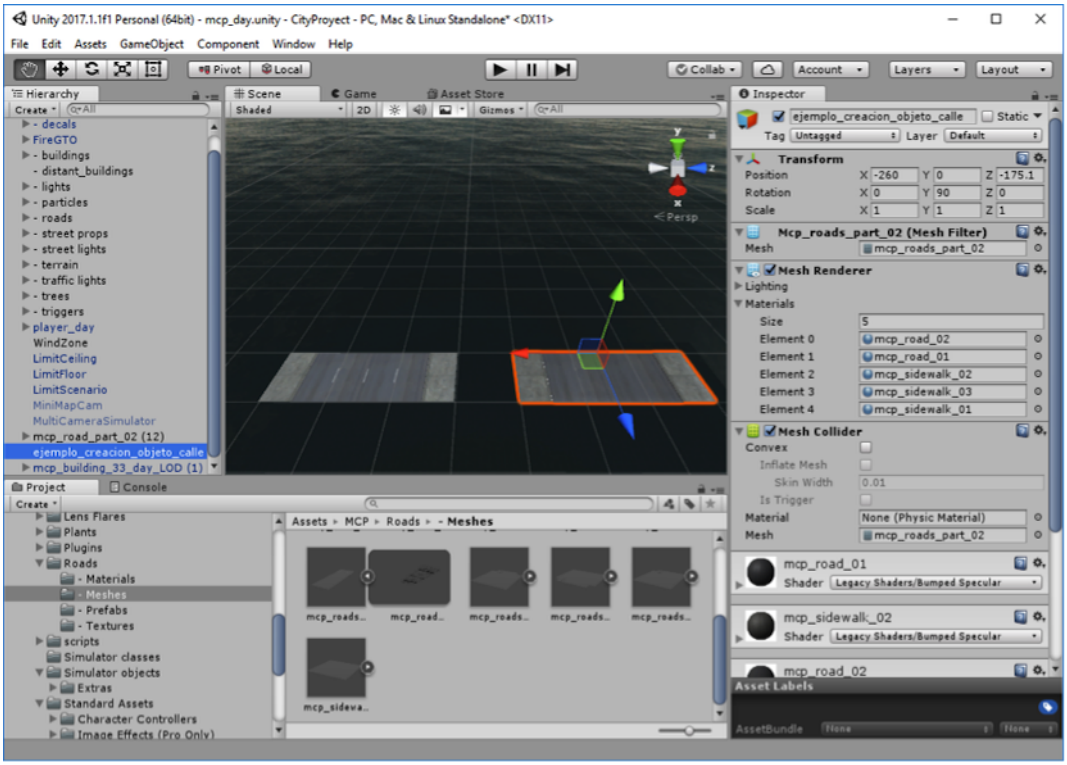


Figura D.7: Paso 5 del tutorial de creación de objeto estático.

Finalmente ya tenemos la calle, podríamos dejarla así, con menor detalle, únicamente con los materiales de las aceras, bordillos y calzada, o si se desea, se le podrían añadir a este objeto, otros objetos estáticos como las líneas blancas que simulan la separación de carriles. Este último paso se haría de una forma similar a la del ejemplo anterior, salvo que no haría falta que las líneas tuvieran el componente *Mesh Collider*, ya que el objeto al que pertenecen ya tiene definida el área o volumen de colisiones.

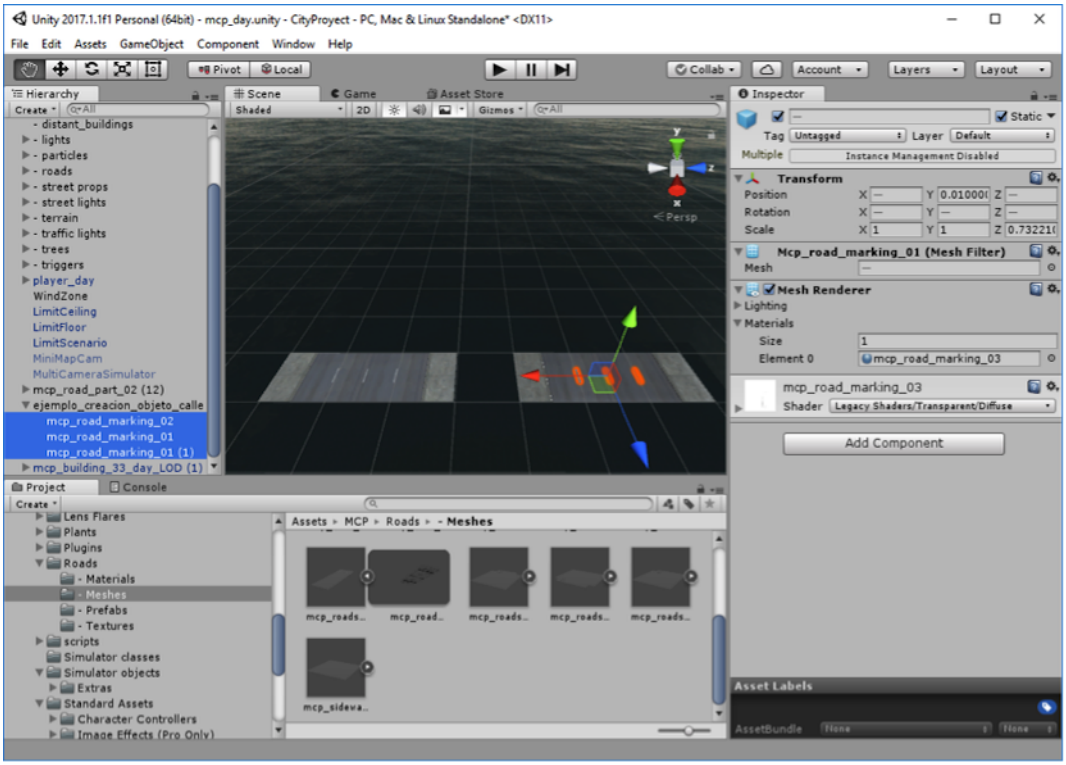


Figura D.8: Paso 6 del tutorial de creación de objeto estático.

D.3. Tutorial de creación de un objeto portátil

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto portátil (**Puerta abatible**) con un *Mesh* preexistente, así como una explicación del *Script* implementado en lenguaje C#. Para más información, puede visitar el link de YouTube[5], donde podrá ver un videotutorial realizando paso por paso, la creación de un objeto portátil.

[5] https://www.youtube.com/watch?v=zgZpwgmDDgM

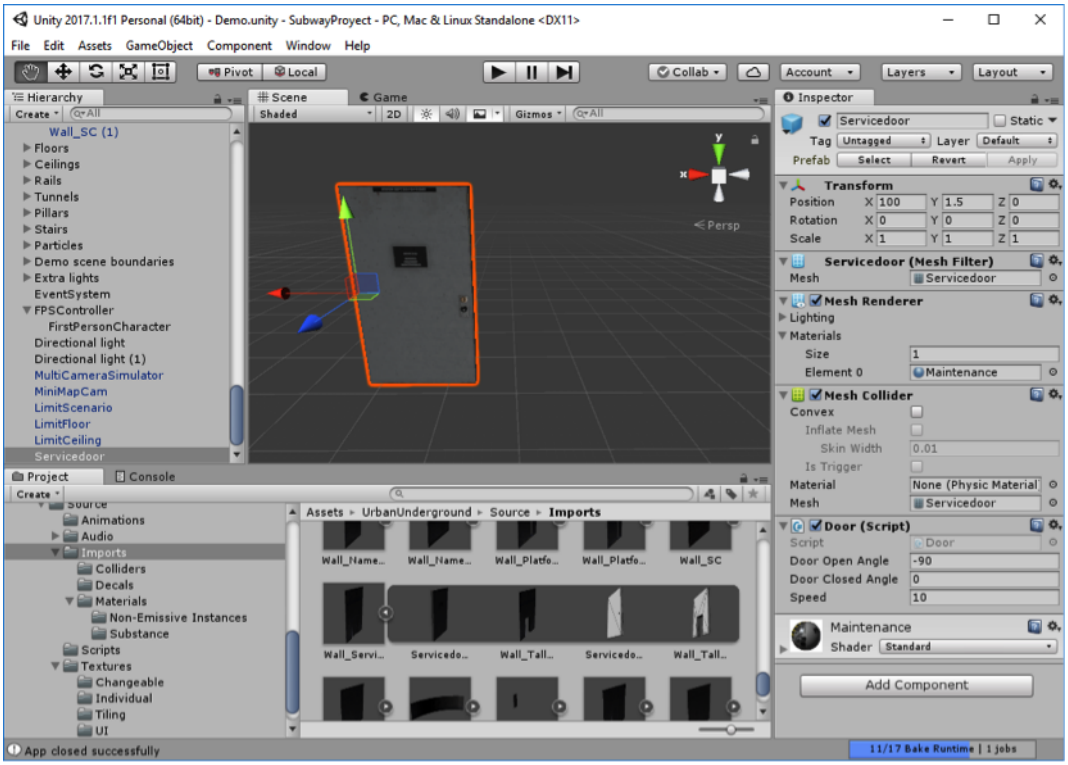


Figura D.9: Paso 1 del tutorial de creación de objeto portátil.

Para crear desde cero el objeto portátil **Puerta abatible** debemos añadir un *GameObject > Create Empty* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto puerta*).

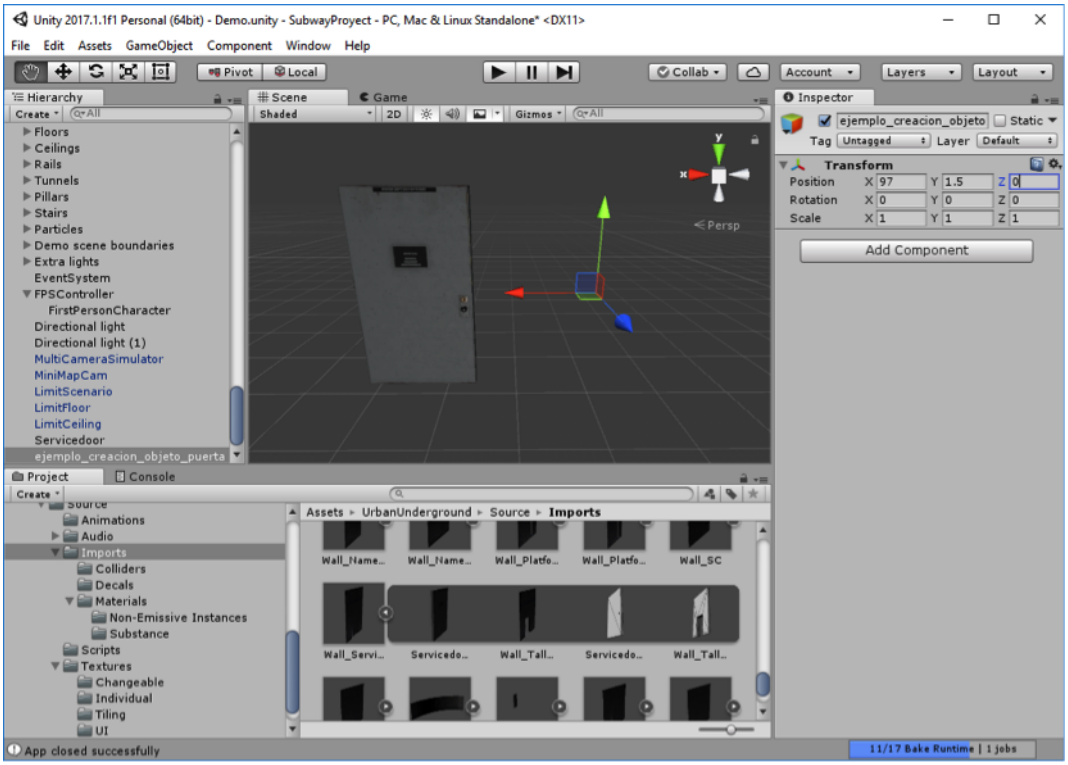


Figura D.10: Paso 2 del tutorial de creación de objeto portátil.

A continuación, se deben añadir al objeto tres componentes más, los cuales cubrirán los elementos que necesita todo objeto portátil, al igual que haríamos con un objeto estático, ya explicado anteriormente:

- **Mesh Filter:** Agrega al objeto el modelo geométrico. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Filter.*
- **Mesh Renderer:** Renderiza el objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Renderer.*
- **Mesh Collider (Box Collider, Sphere Collider o Capsule Collider):** Define el área de colisiones del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Physics > Mesh Collider (Box Collider, Sphere Collider o Capsule Collider).*

Finalmente, se debe añadir al objeto el controlador o *Script* encargado de relacionarse con el objeto dinámico que desee interactuar con el:

- **Script:** Este componente es útil para implementar mediante lenguaje C#, UnityScript o Boo, un componente único y propio, con una funcionalidad y unas características que el usuario requiera en ese momento. Todas las variables que se declaren públicas podrán ser inicializadas desde el *Inspector* o propiamente desde el código fuente del *Script*, accediendo de forma rápida y fácil a las propiedades del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Scripts > (Script a ser cargado. Como por ejemplo: Door).*

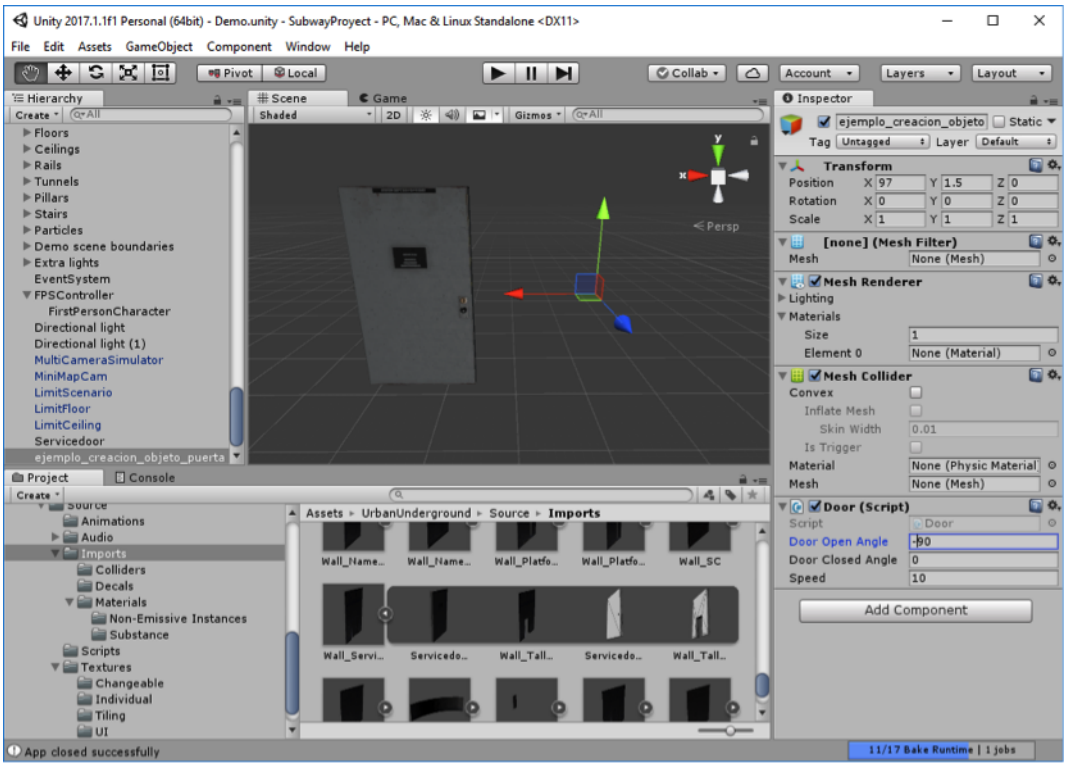


Figura D.11: Paso 3 del tutorial de creación de objeto portátil.

Añadimos tanto en el componente *Mesh Filter*, como en el componente *Mesh Collider* el mismo modelo geométrico ya importado en formato .fbx de ese tipo de puerta (*Servicedoor*).

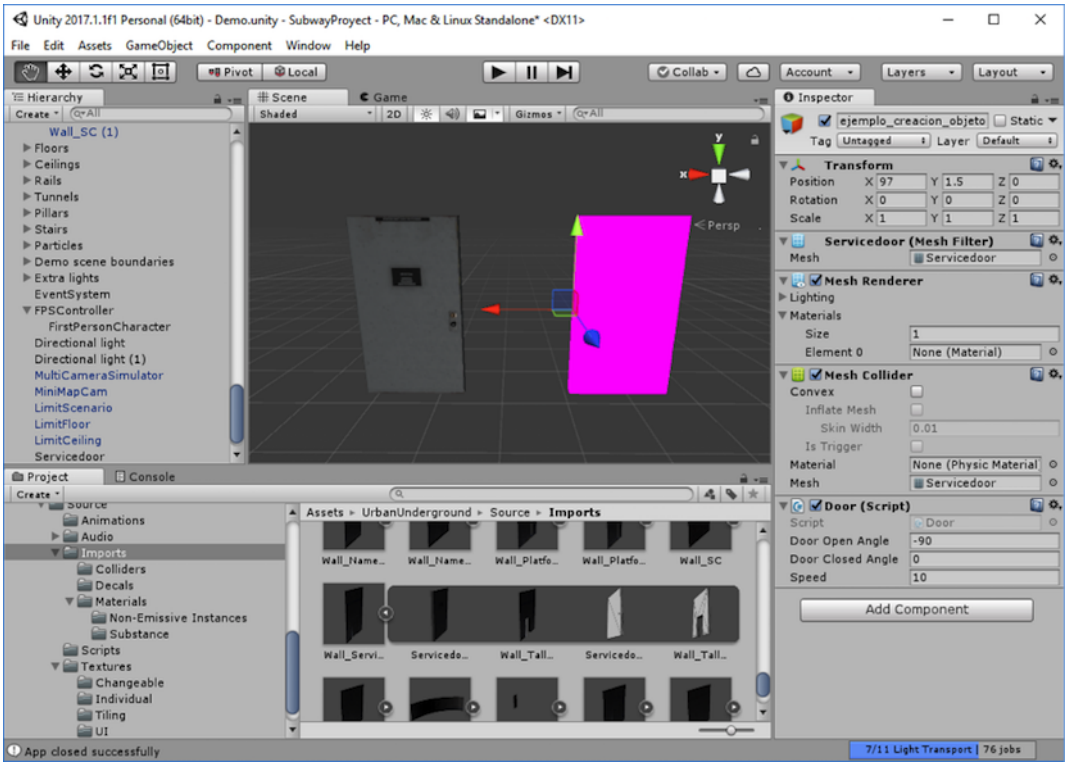


Figura D.12: Paso 4 del tutorial de creación de objeto portátil.

A continuación, para este modelo geométrico que hemos añadido al objeto necesitamos un único material, por lo que solo añadimos el material de la puerta objetivo que queremos obtener (*Maintenance*).

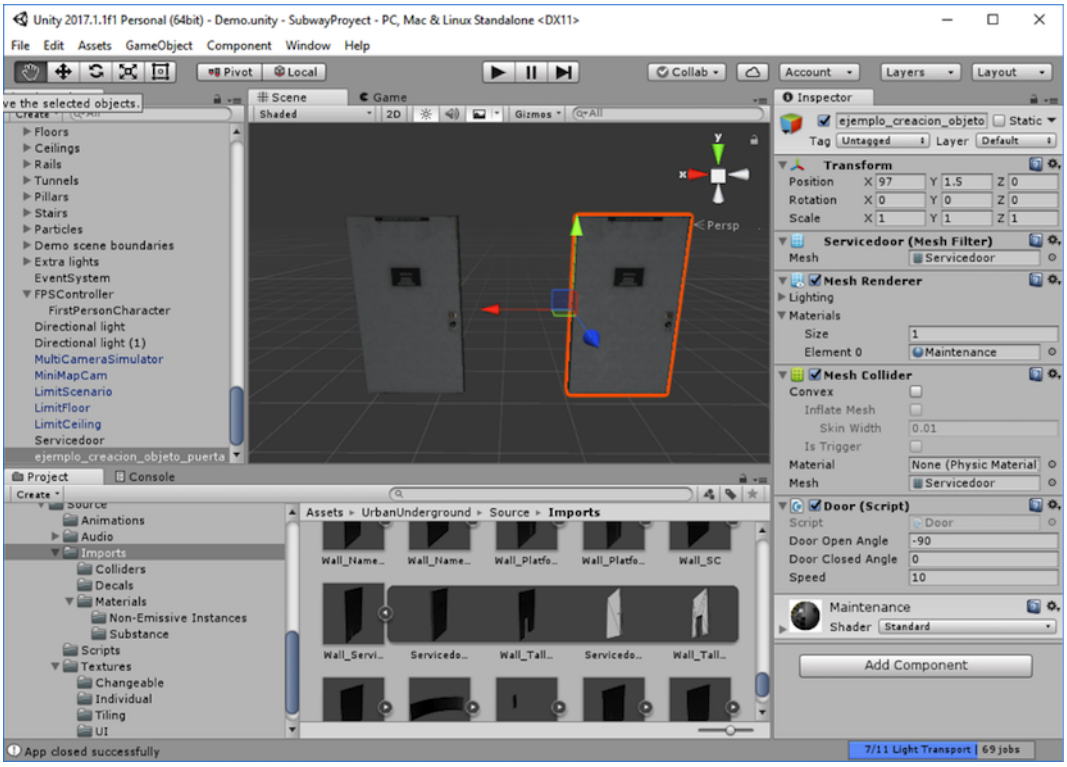


Figura D.13: Paso 5 del tutorial de creación de objeto portátil.

Finalmente, se proporciona el código fuente del *Script* de la puerta llamado *Door*, y una explicación detallada de las variables, métodos y funcionalidad del mismo:

- **Declaración de variables:** Declaramos las variables necesarias para determinar ángulos y el estado de la puerta. Cabe añadir, que todas las variables públicas serán accesibles desde el *Inspector* del objeto portátil, aunque, si no son inicializadas con valores desde el *Inspector* por defecto tomarán los valores definidos.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Door : MonoBehaviour {
5
6     public float doorOpenAngle = 90.0f;
7     public float doorClosedAngle = 0.0f;
8     public float speed = 10.0f;
9     Quaternion doorOpen = Quaternion.identity;
10    Quaternion doorClosed = Quaternion.identity;
11    bool doorStatus = false;
12    bool playerInRange;
13    Transform playerTransform;
14
```

Figura D.14: Ejemplo de declaración de variables del *Script Door*.

- **Método Start():** Este método se ejecutará al inicializarse el objeto, es decir, en el momento en el que el objeto se activa se ejecutará todo el código del interior de este método. En esta función se inicializan los ángulos en los cuales la puerta estará abierta y cerrada, y las propiedades del componente *Transform* del objeto dinámico que interactuará sobre él.

```
15    void Start() {
16        doorOpen = Quaternion.Euler(0, doorOpenAngle, 0);
17        doorClosed = Quaternion.Euler(0, doorClosedAngle, 0);
18        playerTransform = GameObject.FindGameObjectWithTag("Player").transform;
19        return;
20    }
```

Figura D.15: Ejemplo del método Start() del *Script Door.*

- **Método Awake():** Este método se ejecuta al inicializarse el programa, aunque el objeto no se encuentre activo. Es un método opcional, similar al método *Start()*, y ambos sirven para inicializar valores de variables. No suele utilizarse puesto que este método siempre se ejecutará estando o no activo el objeto, al contrario que hacía el método *Start()*, que solo se ejecuta si el objeto esta activo, siendo más eficiente.
- **Método Update():** Este método se ejecutará en cada *frame* de ejecución, por lo que se actualizará y ejecutará todo el código de su interior constantemente. En este método no deberían añadirse inicializaciones, puesto que sobrecargarían la CPU de procesamientos innecesarios en cada *frame*, siendo más óptimo incluirlas en los métodos *Start()* o *Awake()*. En este método se determina si el objeto dinámico se encuentra cerca del objeto portátil, y este puede interactuar con él. En este ejemplo se realiza de forma manual, al recibir el evento de pulsar la tecla 'E'. Esta parte deberá ser automatizada interactuando con un método de otro objeto dinámico que quiera interactuar con él.

```
22    void Update() {
23        if (Vector3.Distance(playerTransform.position, this.transform.position) < 3f) {
24            playerInRange = true;
25
26            if (Input.GetKeyDown(KeyCode.E)) {
27                if (doorStatus) {
28                    StartCoroutine (this.moveDoor(doorClosed));
29                } else {
30                    StartCoroutine (this.moveDoor(doorOpen));
31                }
32            }
33
34        } else {
35            playerInRange = false;
36        }
37        return;
38    }
```

Figura D.16: Ejemplo del método Update() del *Script Door.*

- **Método moveDoor():** Este método evalúa el ángulo de origen que tiene el objeto actualmente y el ángulo de destino al que se desea llegar y realiza dicha rotación con la velocidad indicada hasta que se llegue al ángulo de destino.

```
40    IEnumerator moveDoor(Quaternion target) {
41        while (Quaternion.Angle(transform.localRotation, target) > 0.5f) {
42            transform.localRotation = Quaternion.Slerp (transform.localRotation, target, Time.deltaTime * speed);
43            yield return null;
44        }
45
46        doorStatus = !doorStatus;
47        yield return null;
48    }
49
50 }
```

Figura D.17: Ejemplo del método moveDoor() del *Script Door.*

D.4. Tutorial de creación de un objeto dinámico

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto dinámico (**Coche**) con un *Mesh* y un *prefab* preexistente, así como una explicación del funcionamiento de la ruta que define el movimiento de dicho objeto dinámico. Para más información, puede visitar el link de YouTube[6], donde podrá ver un videotutorial realizando paso por paso, la creación de un objeto dinámico. También puede visitar el siguiente link de YouTube[7] del videotutorial*: Multi-camera System Simulator (MSS) tutorial: Adding a pedestrian*, donde Mario González explicó, paso por paso, como añadir un objeto dinámico persona con una animación de andar o correr.

Se puede observar el objeto dinámico (*FireGTO*), con objetos internos los cuales tienen los componentes necesarios para la creación de dicho objeto (razón por la cual el objeto padre carece de dichos elementos o componentes básicos), así como el *Script* que necesita dicho objeto para realizar el movimiento deseado a la largo de la trayectoria o ruta designada. Se puede observar el *Script* que tiene el objeto ruta del objeto con el que gestiona los puntos y propiedades de dicha ruta.

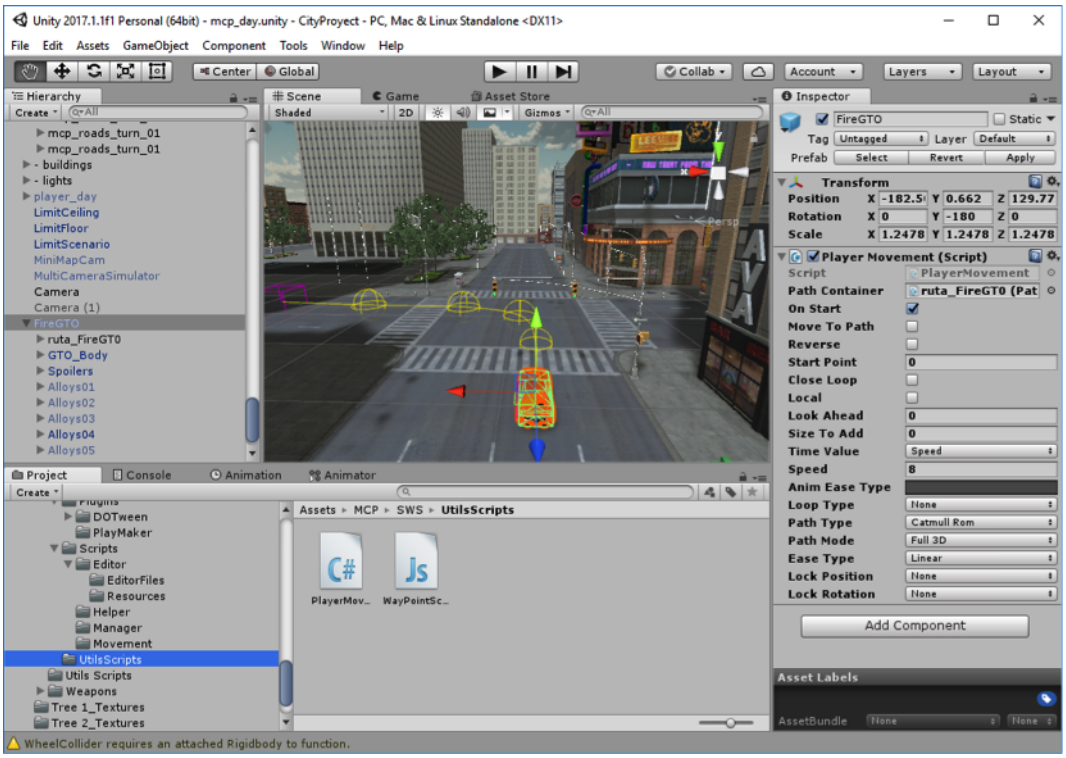


Figura D.18: Paso 1 del tutorial de creación de objeto dinámico.

Se muestra la trayectoria y los puntos que la conforman que tiene asignado el objeto dinámico objetivo (*FireGTO*). Se puede ver como la ruta del objeto es un objeto hijo del objeto dinámico, y los puntos que conforman la ruta, a su vez, son objetos hijos de la propia ruta.

[6] https://www.youtube.com/watch?v=eYyiTtqTwU0
[7] https://www.youtube.com/watch?v=7fXyEiggsmg

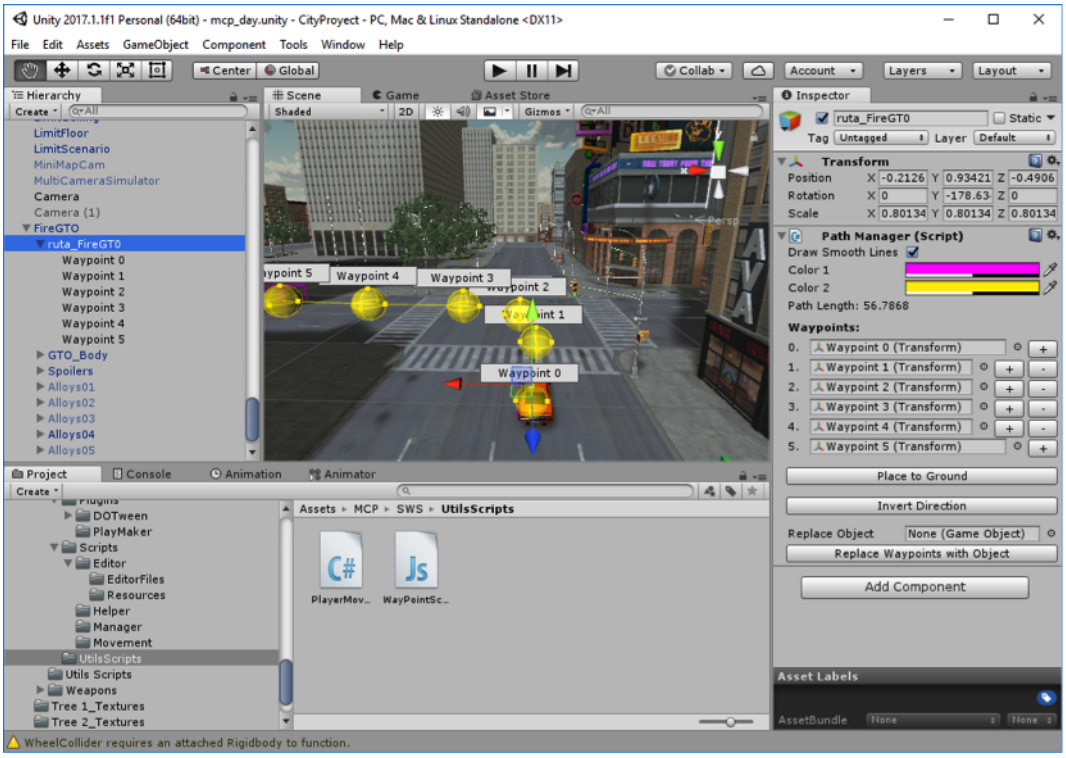


Figura D.19: Paso 2 del tutorial de creación de objeto dinámico.

A continuación, debemos añadir el objeto que queremos hacer dinámico, para posteriormente añadirle una ruta, llamándolo por ejemplo *ejemplo creacion objeto coche*. Se ha añadido un *prefab*, que simula un coche, previamente descargado del Asset Store de Unity, llamado *Cars Free – Muscle Car Pack*[8], arrastrando dicho *prefab* desde la ventana *Project* desde el directorio de origen, hasta la ventana *Scene* en el lugar de destino de la escena.

[8] https://assetstore.unity.com/packages/3d/vehicles/land/cars-free-muscle-car-pack-79854

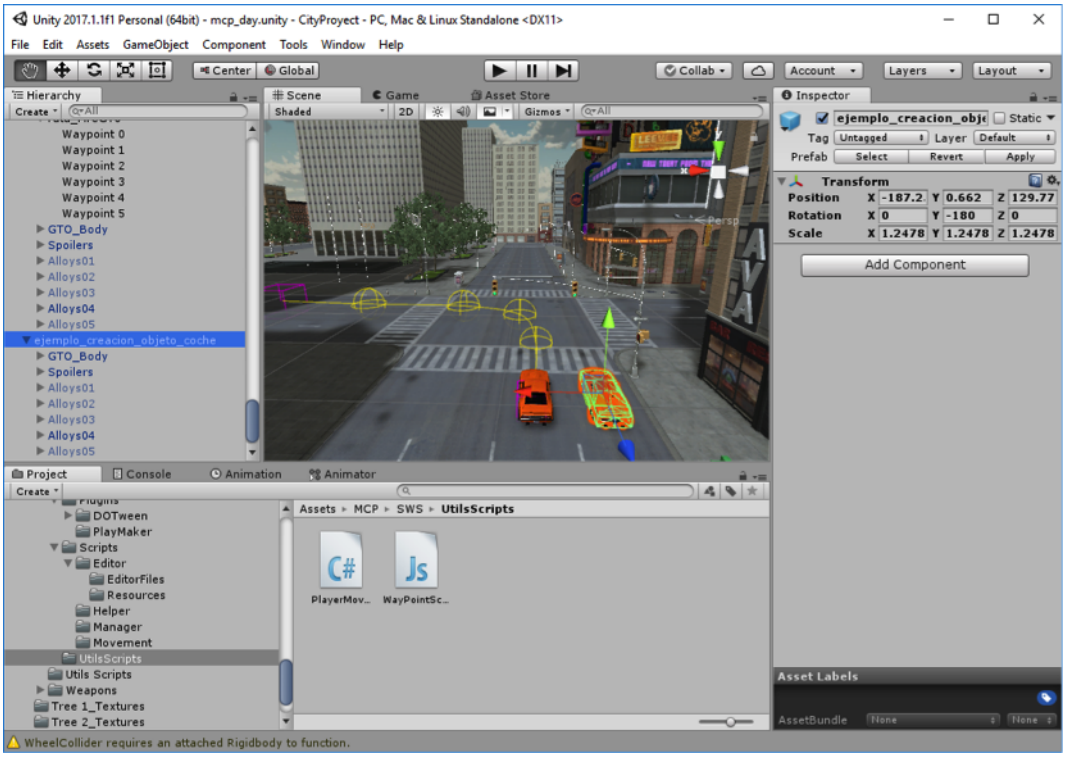


Figura D.20: Paso 3 del tutorial de creación de objeto dinámico.

Al añadir el *prefab* del coche ya tiene añadidos varios objetos hijos con los tres componentes que necesita todo objeto, pero si se crea cualquier otro objeto desde la opción *Create Empty*, se deberán añadir, al igual que haríamos con un objeto estático, ya explicado anteriormente:

- **Mesh Filter:** Agrega al objeto el modelo geométrico. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Filter.*

- **Mesh Renderer:** Renderiza el objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Renderer.*

- **Mesh Collider (Box Collider, Sphere Collider o Capsule Collider):** Define el área de colisiones del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Physics > Mesh Collider (Box Collider, Sphere Collider o Capsule Collider).*

Seguidamente, se debe añadir al objeto dinámico el controlador o *Script* encargado de mover y girar (modificando su posición y rotación) del propio objeto a lo largo de una ruta o trayecto definido:

- **Script:** Este componente es útil para implementar mediante lenguaje C#, UnityScript o Boo, un componente único y propio, con una funcionalidad y unas características que el usuario requiera en ese momento. Todas las variables que se declaren públicas podrán ser inicializadas desde el *Inspector* o propiamente desde el código fuente del *Script*, accediendo de forma rápida y fácil a las propiedades del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Scripts > (Script a ser cargado. Como por ejemplo: Player Movement).*

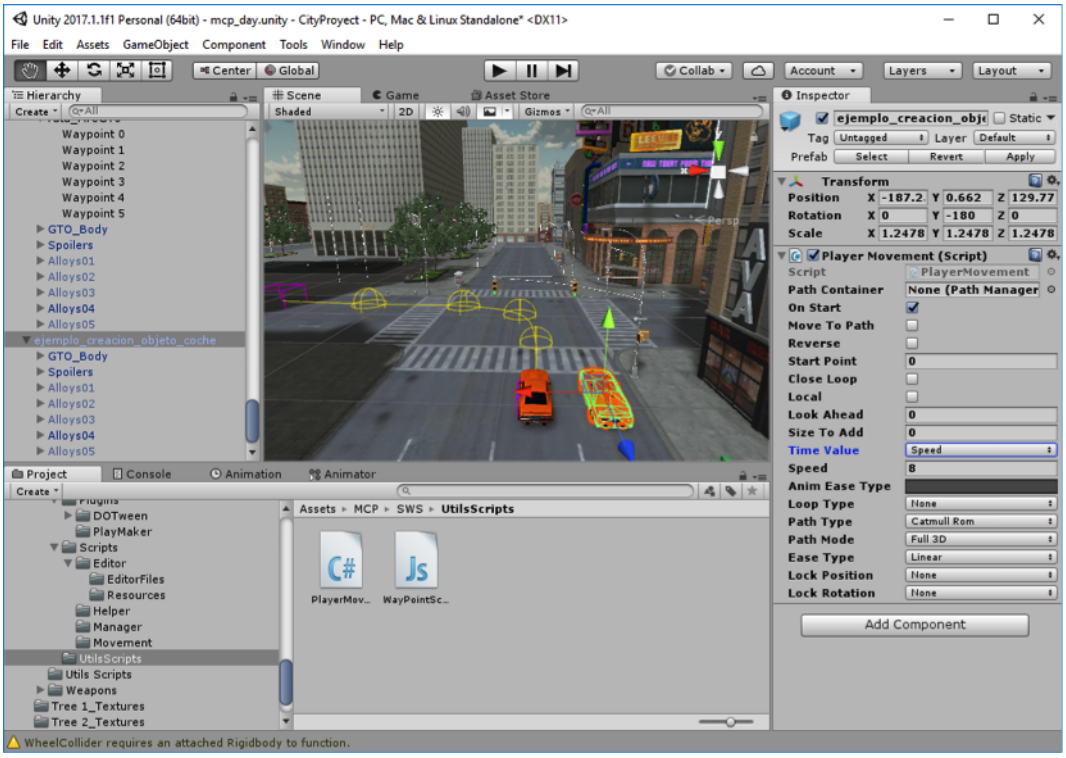


Figura D.21: Paso 4 del tutorial de creación de objeto dinámico.

Por ahora dejamos el campo *Path Container* vacío, sin ninguna ruta. Pero una vez creada, se deberá añadir aquí el objeto ruta que deseemos que siga nuestro objeto dinámico, y modificar las propiedades por si queremos que se repita cíclicamente por ejemplo. Para crear rutas y recorrido fácilmente, debemos importar la funcionalidad descargada del Asset Store de Unity, llamado *Simple Waypoint System* (SWS)[9], para la creación de la ruta que realizará el objeto y añadir un gestor para la creación de dicha trayectoria. Primero debemos añadir un *GameObject > Create Empty* y llamarlo (por ejemplo: *Manager-SWS*). Una vez creado este objeto vacío, le tenemos que añadir un *Script: Add Component > Scripts > SWS > Waypoint Manager*, el cual nos facilita notablemente la creación de rutas y trayectorias.

[9] https://assetstore.unity.com/packages/tools/animation/simple-waypoint-system-2506

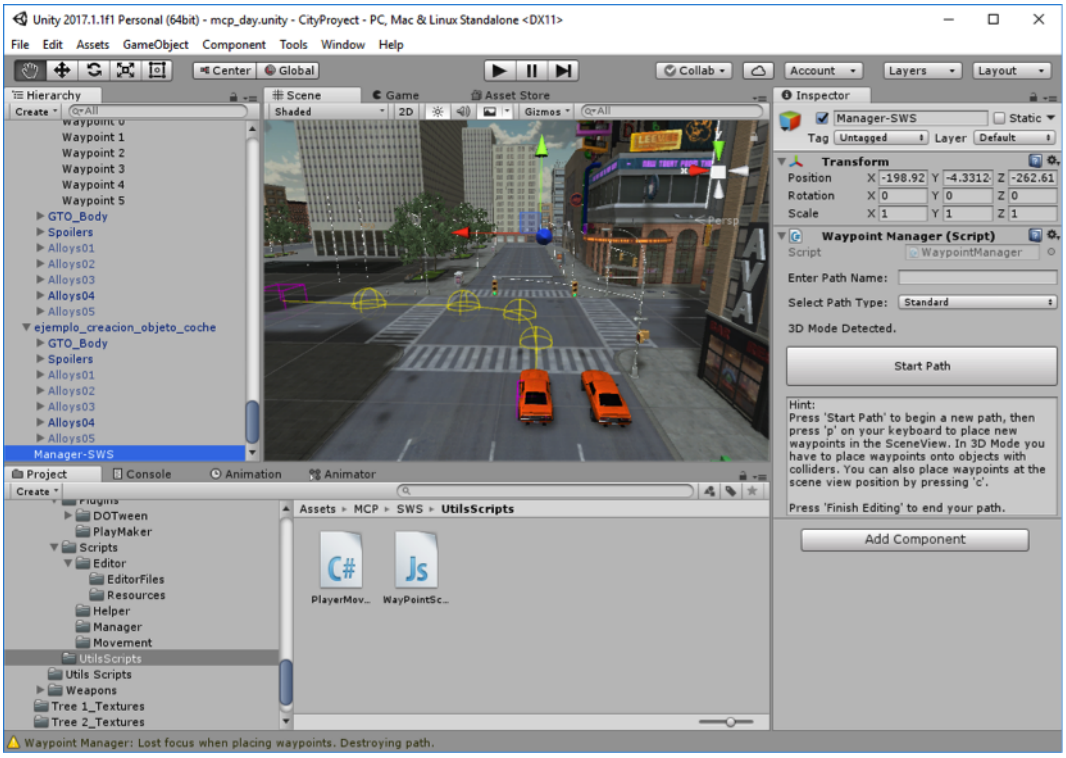


Figura D.22: Paso 5 del tutorial de creación de objeto dinámico.

Seguidamente, con la ayuda del *Script* que acabamos de añadir al gestor podremos crear fácilmente la ruta, asignándole un nombre a la ruta (por ejemplo: *ruta ejemplo creacion objeto coche*), y pulsando sobre el botón *Start Path*. Con la tecla 'P' podremos añadir puntos a la ruta (también llamados *Waypoints*). Una vez finalizada la ruta pulsamos sobre el mismo botón, ahora llamado *Finish Editing*.

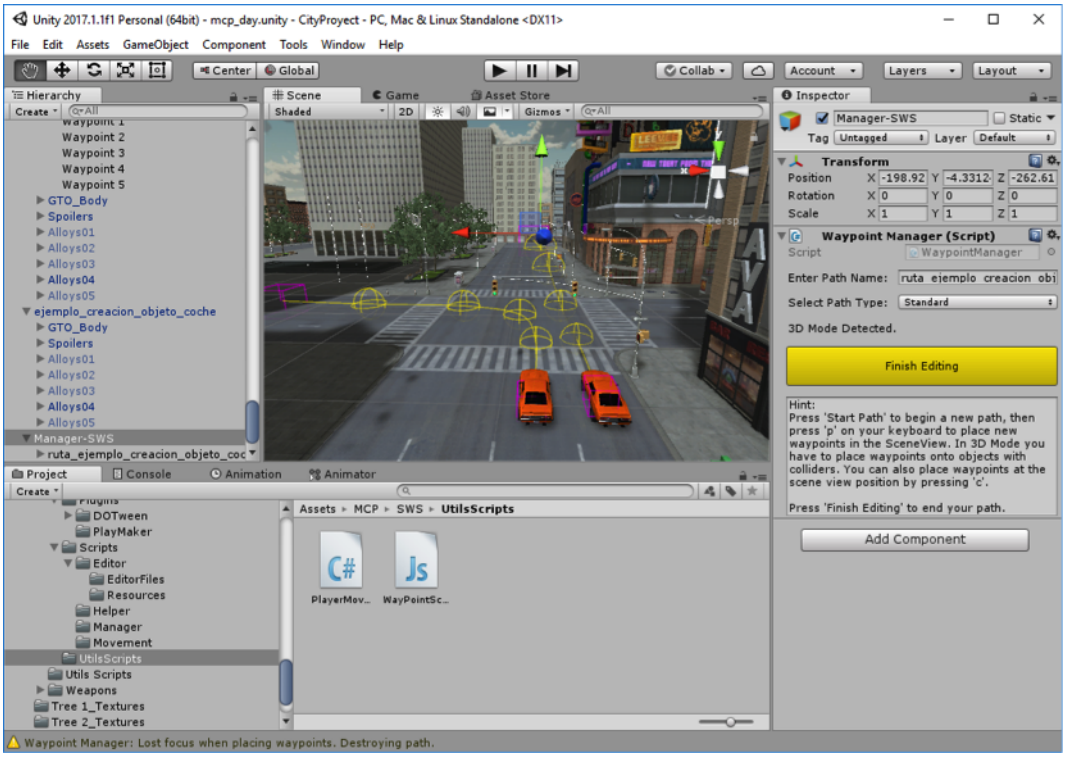


Figura D.23: Paso 6 del tutorial de creación de objeto dinámico.

Se puede ver como se ha creado la ruta con el nombre *ruta ejemplo creacion objeto coche*, junto con un *Script* que se ha creado automáticamente llamado *Path Manager,* en el cual se pueden apreciar los puntos (*Waypoints*) que conforman la ruta que añadimos anteriormente, facilitándonos la tarea de añadir nuevos puntos de la ruta, o de borrar algunos puntos existentes.

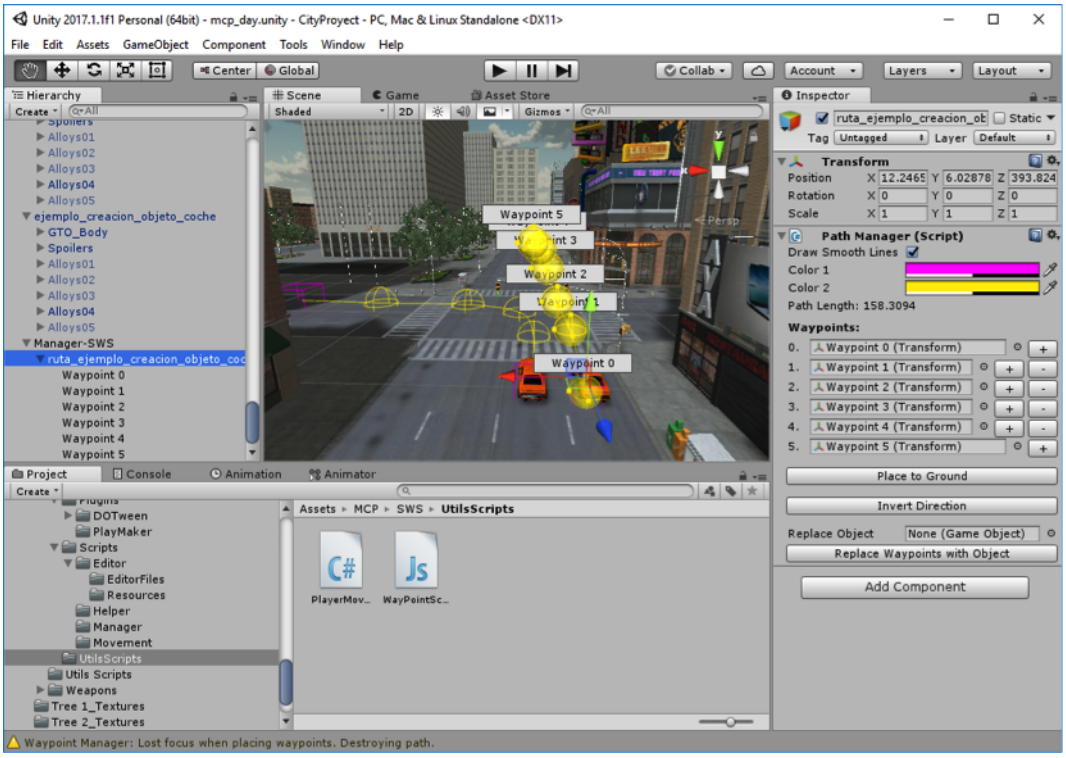


Figura D.24: Paso 7 del tutorial de creación de objeto dinámico.

Finalmente, añadimos la ruta creada (*ruta ejemplo creacion objeto coche*), como objeto hijo de nuestro objeto dinámico (*ejemplo creacion objeto coche*) y añadimos la ruta que hemos creado en el campo *Path Container* para referenciar al *Script* del movimiento del objeto, por la trayectoria que debe desplazarse mientras modifica su posición y su rotación para alcanzar el objetivo.

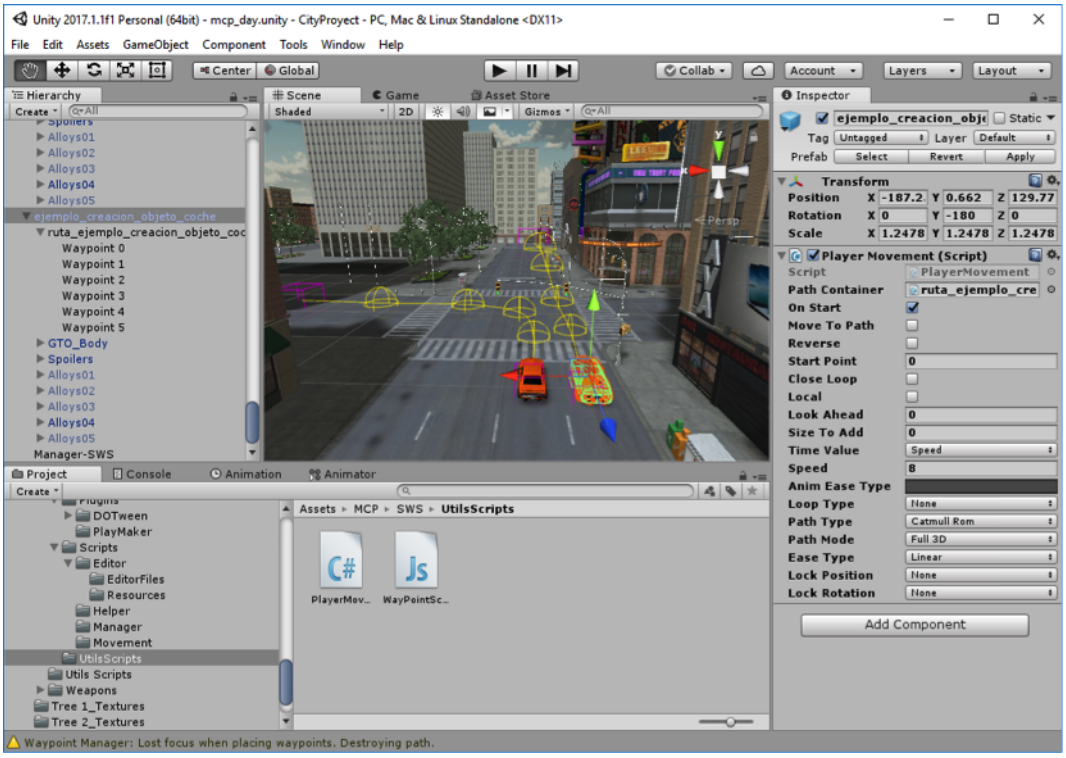


Figura D.25: Paso 8 del tutorial de creación de objeto dinámico.

Una vez finalizada la creación del objeto dinámico con los componentes y elementos mínimos que debe tener, se podrían añadir elementos opcionales, como una animación, normalmente controlada con un *Animator* o *Animation* (como se muestra en el tutorial del objeto periódico), que indique una transición de estados y de distintos movimientos que realizaría, o como un controlador o *Script* encargado de interactuar con los *Scripts* de los objetos portátiles, con el fin de modificar la posición o rotación del objeto portátil.

D.5. Tutoriales de creación de los objetos periódicos

Para más información, puede visitar el link de YouTube[10], donde podrá ver un videotutorial realizando paso por paso, las creaciones de los distintos objetos periódicos.

D.5.1. Objeto periódico corpóreo con Script

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto periódico corpóreo con *Script* (**Pantalla publicitaria**) con un *Mesh* preexistente, así como una explicación del *Script* implementado en lenguaje UnityScript.

[10] https://www.youtube.com/watch?v=3QOIuizvy2c

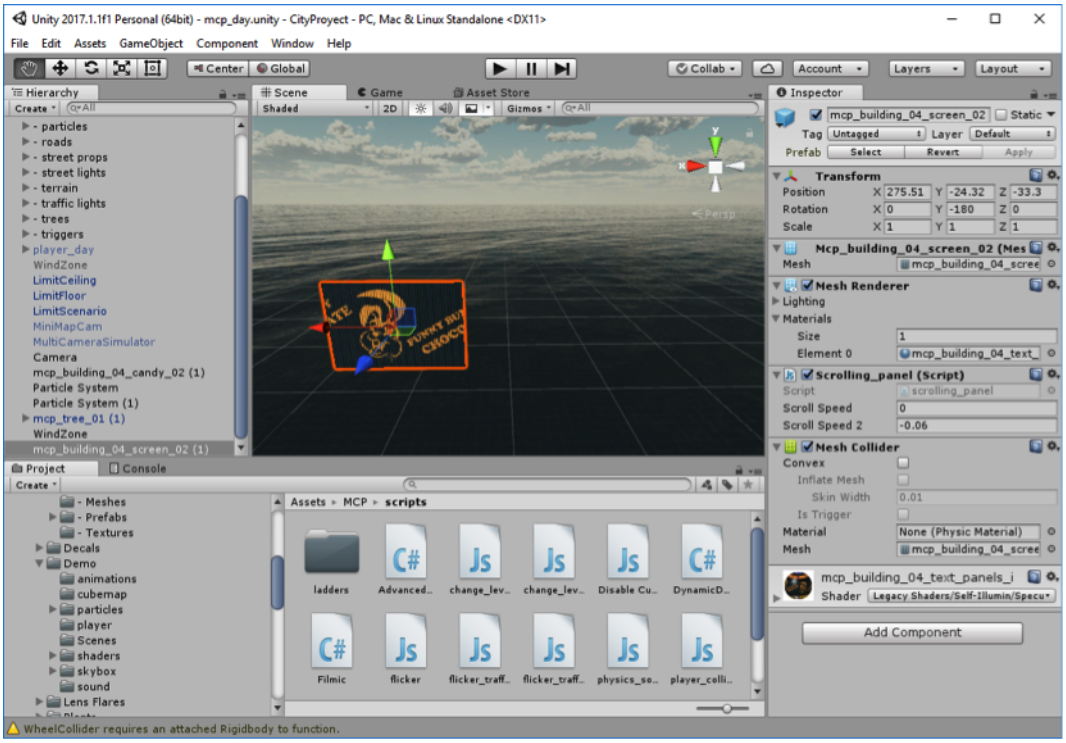


Figura D.26: Paso 1 del tutorial de creación de objeto periódico corpóreo con *Script*.

Para crear desde cero el objeto periódico **Pantalla publicitaria** debemos añadir un *GameObject > Create Empty* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto pantalla*).

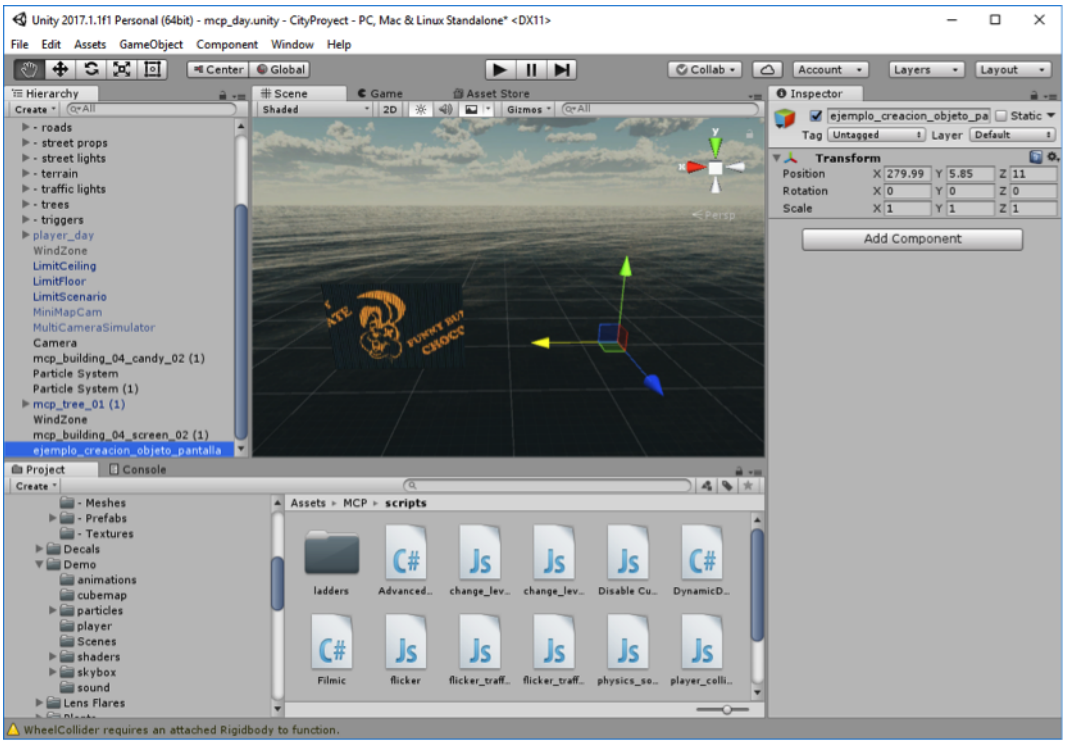


Figura D.27: Paso 2 del tutorial de creación de objeto periódico corpóreo con *Script*.

A continuación, se deben añadir al objeto tres componentes más, los cuales necesita todo objeto periódico de modelo 1, al igual que haríamos con un objeto estático, ya explicado anteriormente:

- **Mesh Filter:** Agrega al objeto el modelo geométrico. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Filter.*
- **Mesh Renderer:** Renderiza el objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Renderer.*
- **Mesh Collider (Box Collider, Sphere Collider o Capsule Collider):** Define el área de colisiones del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Physics > Mesh Collider (Box Collider, Sphere Collider o Capsule Collider).*

Finalmente, se debe añadir al objeto el controlador o *Script* encargado de realizar alguna modificación sobre la posición o rotación del propio objeto de forma periódica:

- **Script:** Este componente es útil para implementar mediante lenguaje C#, UnityScript o Boo, un componente único y propio, con una funcionalidad y unas características que el usuario requiera en ese momento. Todas las variables que se declaren públicas podrán ser inicializadas desde el *Inspector* o propiamente desde el código fuente del *Script*, accediendo de forma rápida y fácil a las propiedades del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Scripts > (Script a ser cargado. Como por ejemplo: Scrolling panel).*

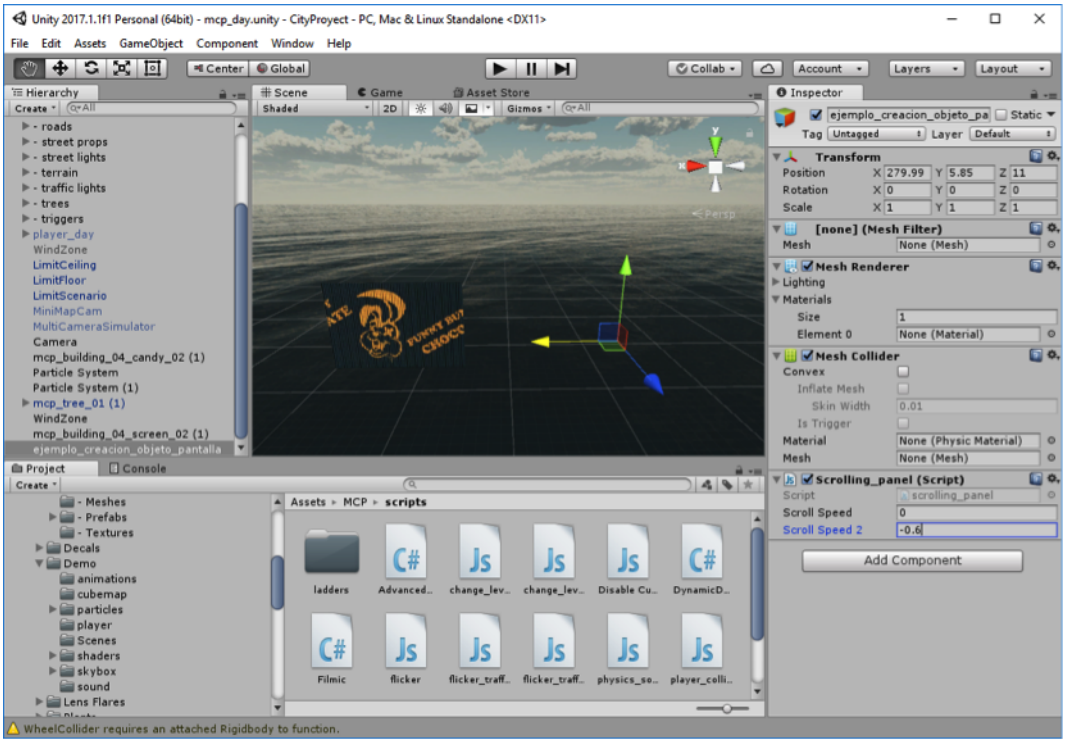


Figura D.28: Paso 3 del tutorial de creación de objeto periódico corpóreo con *Script.*

Añadimos tanto en los componentes *Mesh Filter* y *Mesh Collider* el mismo modelo geométrico ya importado en formato .fbx de ese tipo de pantalla publicitaria (*mcp building 04 screen 02*). A continuación, para este modelo geométrico que hemos añadido al objeto nece-

sitamos un único material, por lo que solo añadimos el material de la pantalla publicitaria objetivo que queremos obtener (*mcp building 04 text panels i*).

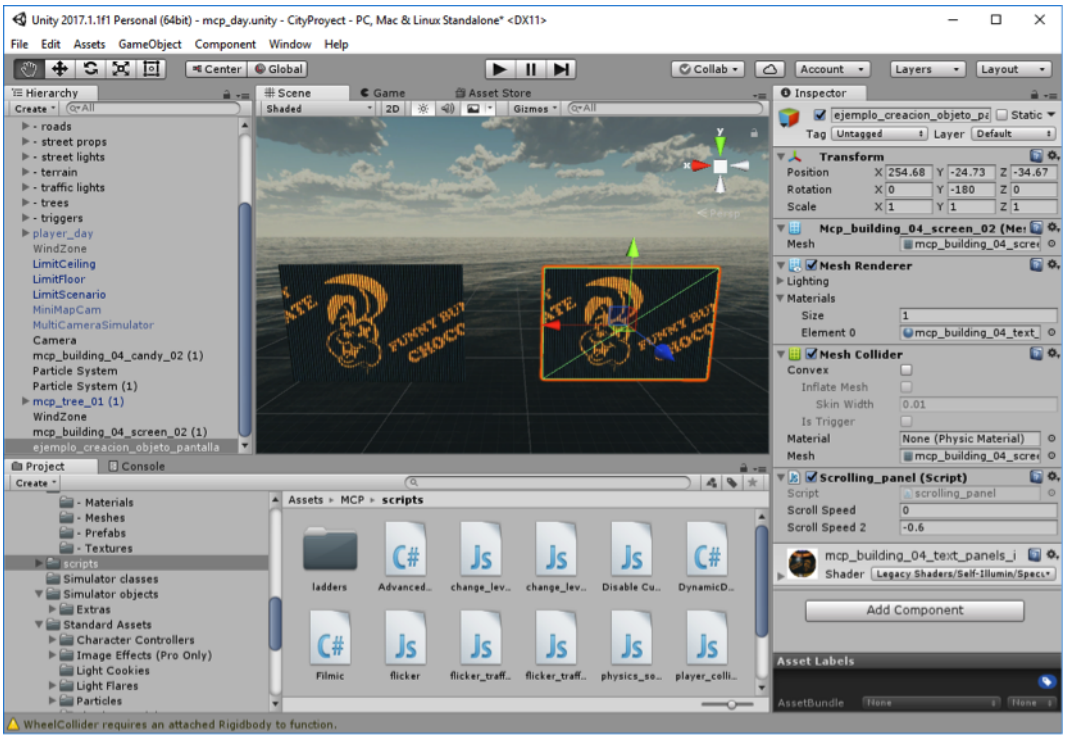


Figura D.29: Paso 4 del tutorial de creación de objeto periódico corpóreo con *Script*.

Finalmente, se proporciona el código fuente del *Script* de la pantalla publicitaria llamado *Scrolling panel*, y una explicación detallada de las variables, métodos y funcionalidad del mismo:

- **Declaración de variables:** Declaramos las variables necesarias para la velocidad de movimiento de la pantalla publicitaria. Cabe añadir, que todas las variables públicas serán accesibles desde el *Inspector* del objeto portátil, aunque, si no son inicializadas con valores desde el *Inspector* por defecto tomarán los valores definidos.

```
1 var scrollSpeed = 0.90;
2
3 var scrollSpeed2 = 0.90;
4
```

Figura D.30: Ejemplo de declaración de variables del *Script Scrolling panel*.

- **Método Update():** Este método se ejecutará en cada *frame* de ejecución, por lo que se actualizará y ejecutará todo el código de su interior constantemente. Este método desplaza la renderización del material en cada *frame*, para dar una sensación de rotación o periodicidad al movimiento, siendo siempre continuo.

```
function Update() {

    var offset = Time.time * scrollSpeed;

    var offset2 = Time.time * scrollSpeed2;

    GetComponent.<Renderer>().material.mainTextureOffset = Vector2 (offset2,-offset);

}
```

Figura D.31: Ejemplo del método Update() del *Script Scrolling panel.*

D.5.2. Objeto periódico corpóreo con animación

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto periódico corpóreo con animación (**Publicidad rotacional**) con un *Mesh* preexistente, así como una explicación de la creación de animaciones:

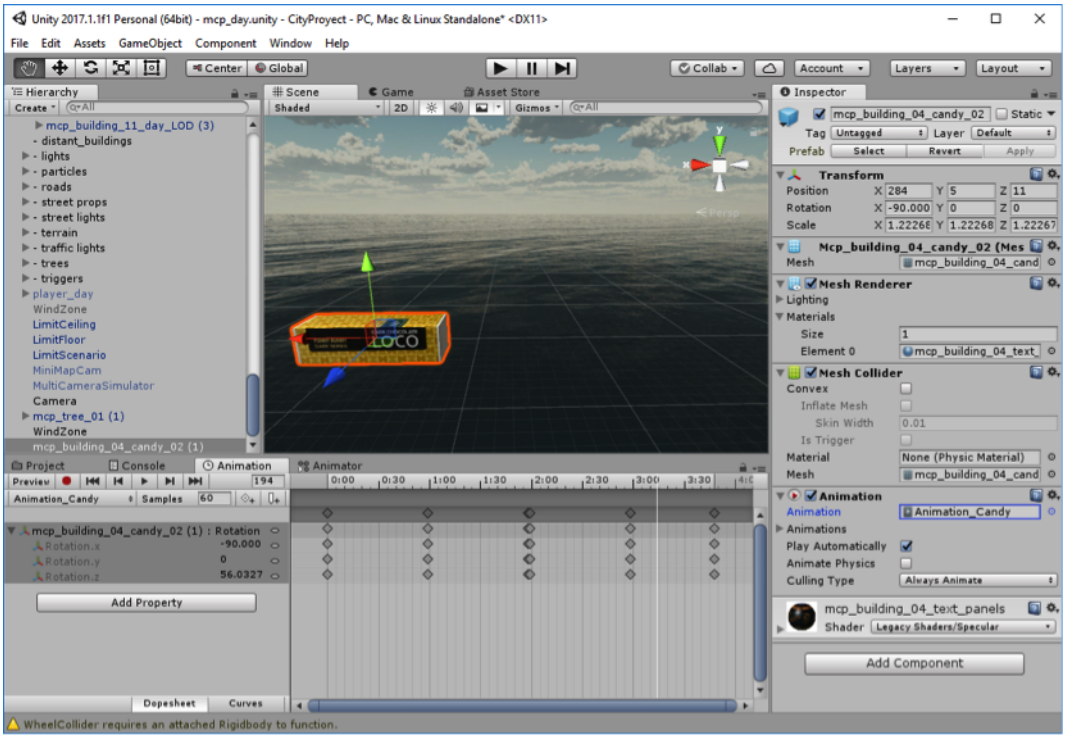


Figura D.32: Paso 1 del tutorial de creación de objeto periódico corpóreo con animación.

Para crear desde cero el objeto periódico **Publicidad rotacional** debemos añadir un *GameObject > Create Empty* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto publicidad*).

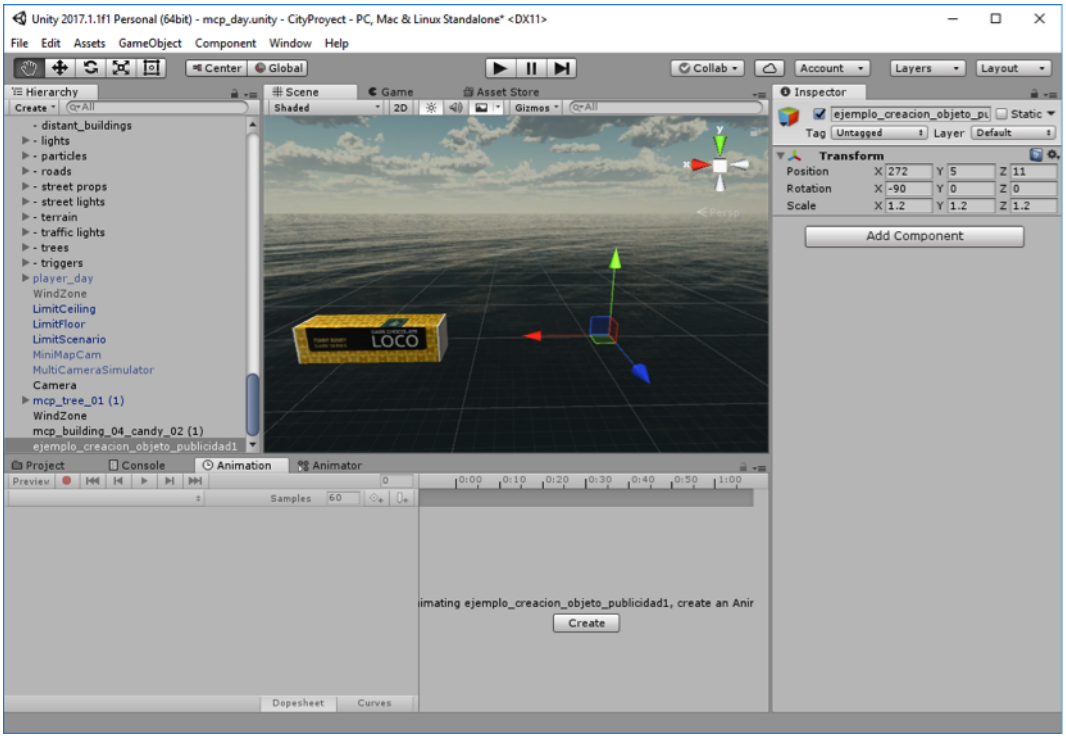


Figura D.33: Paso 2 del tutorial de creación de objeto periódico corpóreo con animación.

A continuación, se deben añadir al objeto tres componentes más, los cuales necesita todo objeto periódico de modelo 1, al igual que haríamos con un objeto estático, ya explicado anteriormente:

- **Mesh Filter:** Agrega al objeto el modelo geométrico. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Filter.*

- **Mesh Renderer:** Renderiza el objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Mesh > Mesh Renderer.*

- **Mesh Collider (Box Collider, Sphere Collider o Capsule Collider):** Define el área de colisiones del objeto. De la siguiente forma podrá añadir este componente el objeto: *Add Component > Physics > Mesh Collider (Box Collider, Sphere Collider o Capsule Collider).*

Finalmente, se debe añadir al objeto la animación encargada de realizar alguna modificación sobre la posición o rotación del propio objeto de forma periódica:

- **Animation/Animator:** Este componente es útil para otorgar al objeto una animación ya creada o definida, la cual le atribuya ciertas modificaciones sobre su posición o su rotación. Dicha animación se podría configurar de forma que se repita constantemente, configurándola como un bucle (*Loop*). De la siguiente forma podrá añadir este componente el objeto: *Add Component > Miscellaneous > Animation/Animator*

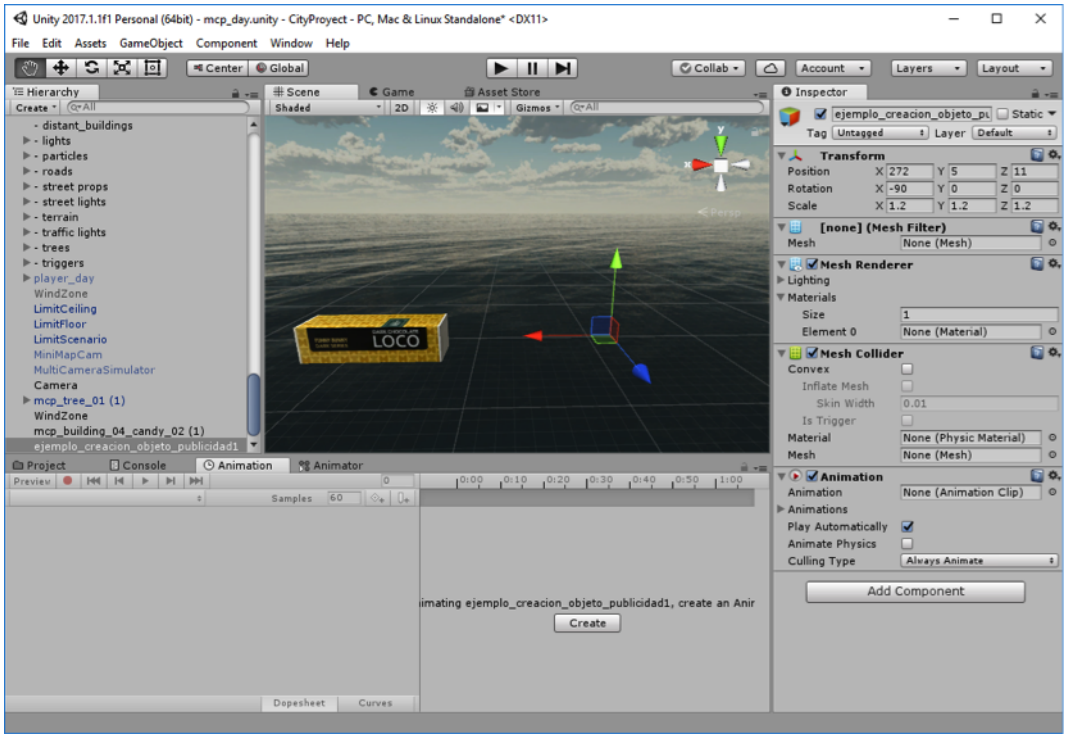


Figura D.34: Paso 3 del tutorial de creación de objeto periódico corpóreo con animación.

Añadimos tanto en los componentes *Mesh Filter* y *Mesh Collider* el mismo modelo geométrico ya importado en formato .fbx de ese tipo de pantalla publicitaria (*mcp building 04 candy 02*). A continuación, para este modelo geométrico que hemos añadido al objeto necesitamos un único material, por lo que solo añadimos el material de la pantalla publicitaria objetivo que queremos obtener (*mcp building 04 text panels*).

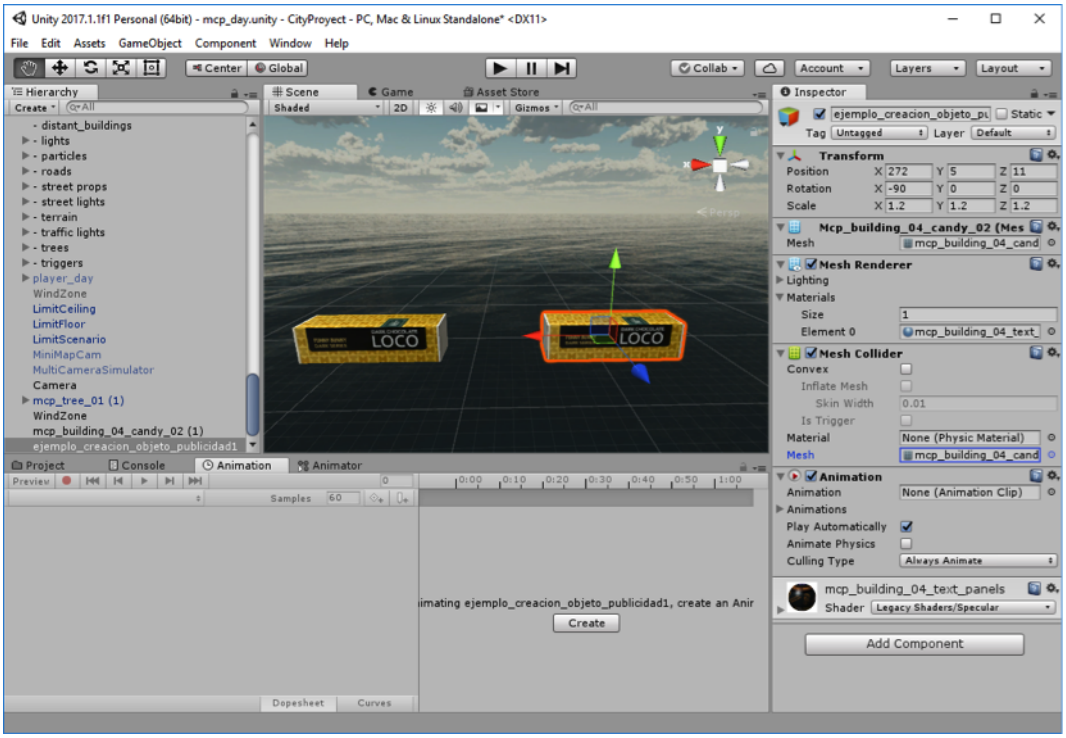


Figura D.35: Paso 4 del tutorial de creación de objeto periódico corpóreo con animación.

Finalmente, para crear una animación sencilla, la cual rote indefinidamente el objeto que hemos creado, deberemos abrir la ventana de *Animation* (la cual puede verse en la parte inferior de la imagen, si no está abierta, se puede abrir pulsando sobre el desplegable *Window* de la barra de menús), y a continuación, pulsar sobre el botón *Create* de la ventana *Animation*, para empezar a crear nuestra animación.

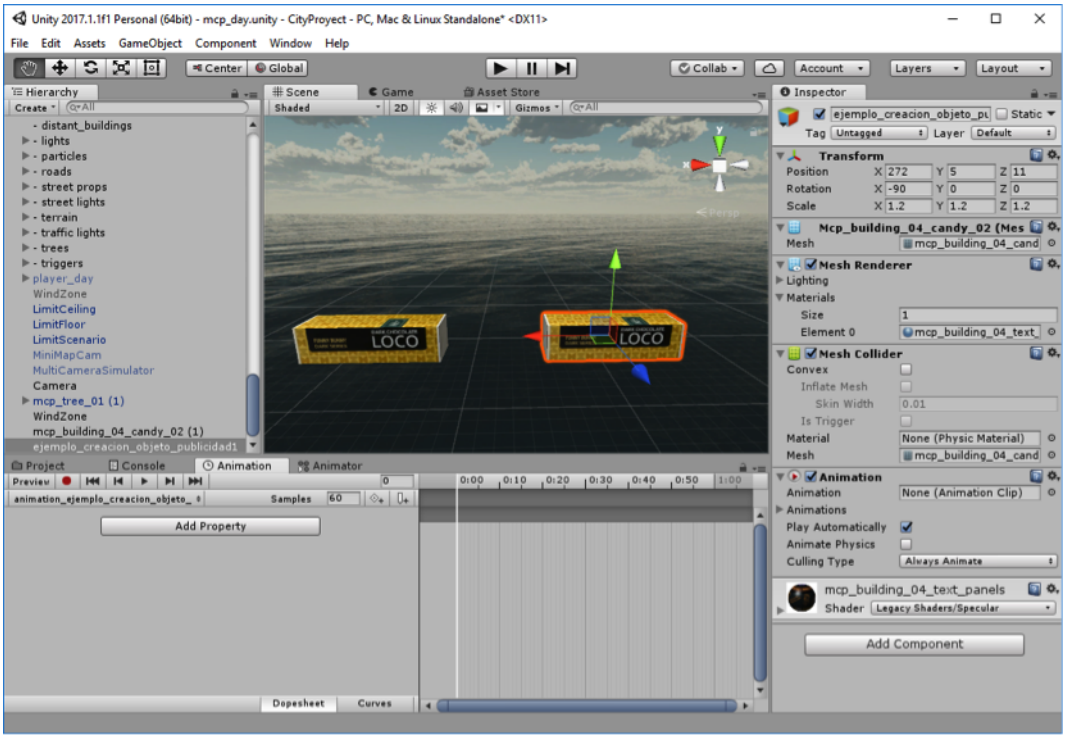


Figura D.36: Paso 5 del tutorial de creación de objeto periódico corpóreo con animación.

Tras crear la animación, añadimos la propiedad que deseamos modificar a lo largo de la animación, mediante el botón *Add Property* de la ventana *Animation*, y añadimos la propiedad deseada. En nuestro caso, hemos querido modificar la propiedad *Rotación* del componente *Transform* del objeto.

Añadimos puntos con distintos valores para la rotación en el eje que deseemos (por ejemplo el eje z), a distintos tiempos. Automáticamente, la animación hará transiciones entre los distintos valores de rotación en cada tiempo que se haya definido.

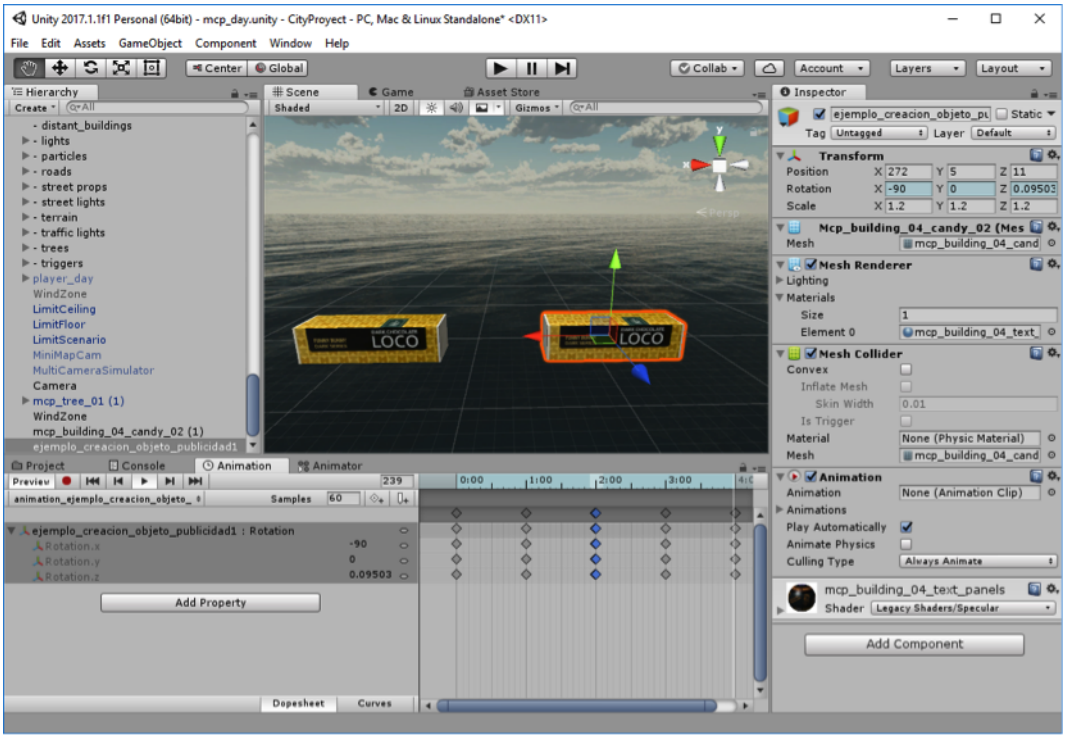


Figura D.37: Paso 6 del tutorial de creación de objeto periódico corpóreo con animación.

Cabe añadir, que para configurar la animación como un bucle continuo, se deberá pinchar sobre la animación que hayamos creado, y nos aparecerá una ventana como la siguiente en el *Inspector*, y en el desplegable *Wrap Mode*, cambiar el modo a *Loop*, o al modo que mejor convenga en cada situación.

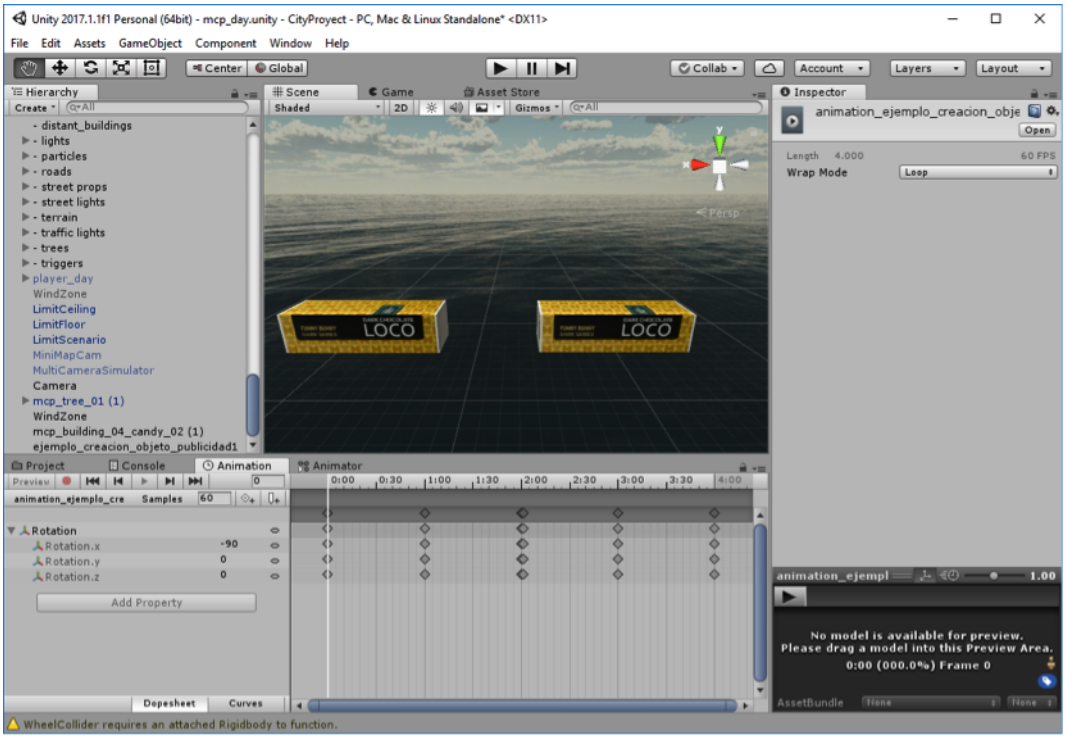


Figura D.38: Paso 7 del tutorial de creación de objeto periódico corpóreo con animación.

D.5.3. Objeto periódico corpóreo (Árbol con viento)

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto periódico corpóreo (Árbol con viento) con un *Mesh* preexistente (Objetos primitivos de Unity: *Tree* y *Wind Zone*). Los parámetros de edición del árbol podrán ser modificados a gusto del usuario, e incluso existen herramientas de creación procedural de este tipo de objetos:

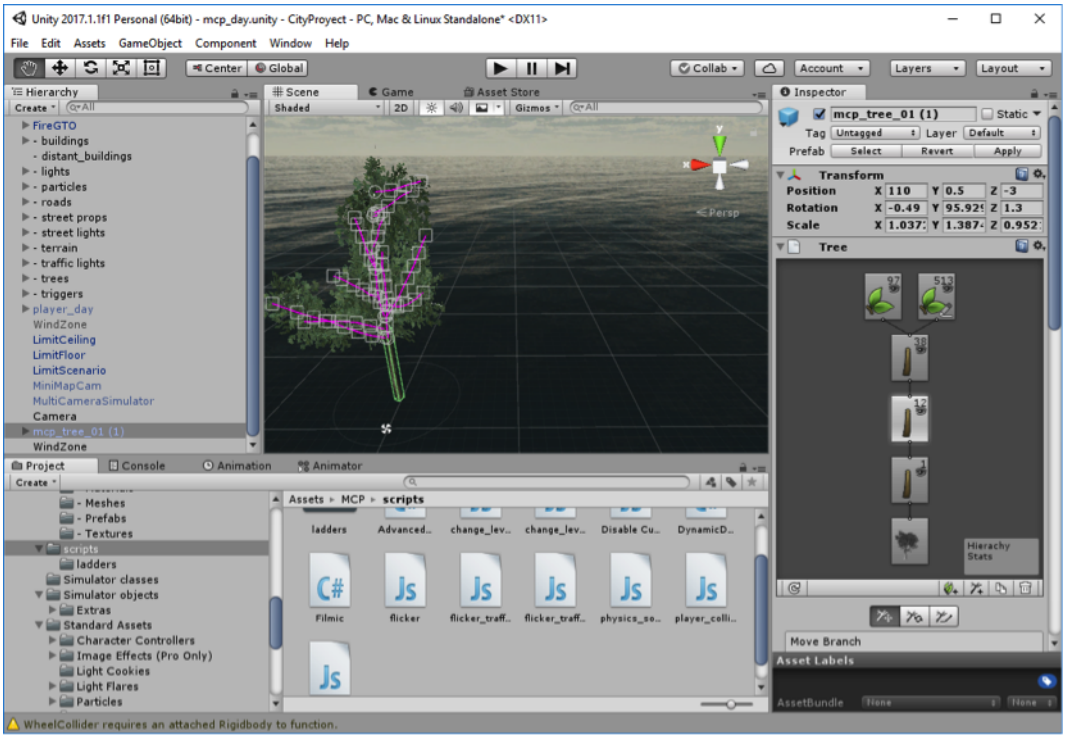


Figura D.39: Paso 1 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

Para crear desde cero el objeto periódico **Árbol con viento** debemos añadir un *GameObject > 3D Object > Tree* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto arbol*). En la ventana *Inspector* puede verse la herramienta de edición del objeto árbol, con las distintas propiedades y configuraciones que se puede aplicar a cada parte del árbol, además, se puede ver de forma sencilla la jerarquía de cada elemento del árbol. Al crear el objeto *Tree*, viene creado por defecto una rama o tronco raíz, con varios valores por defecto.

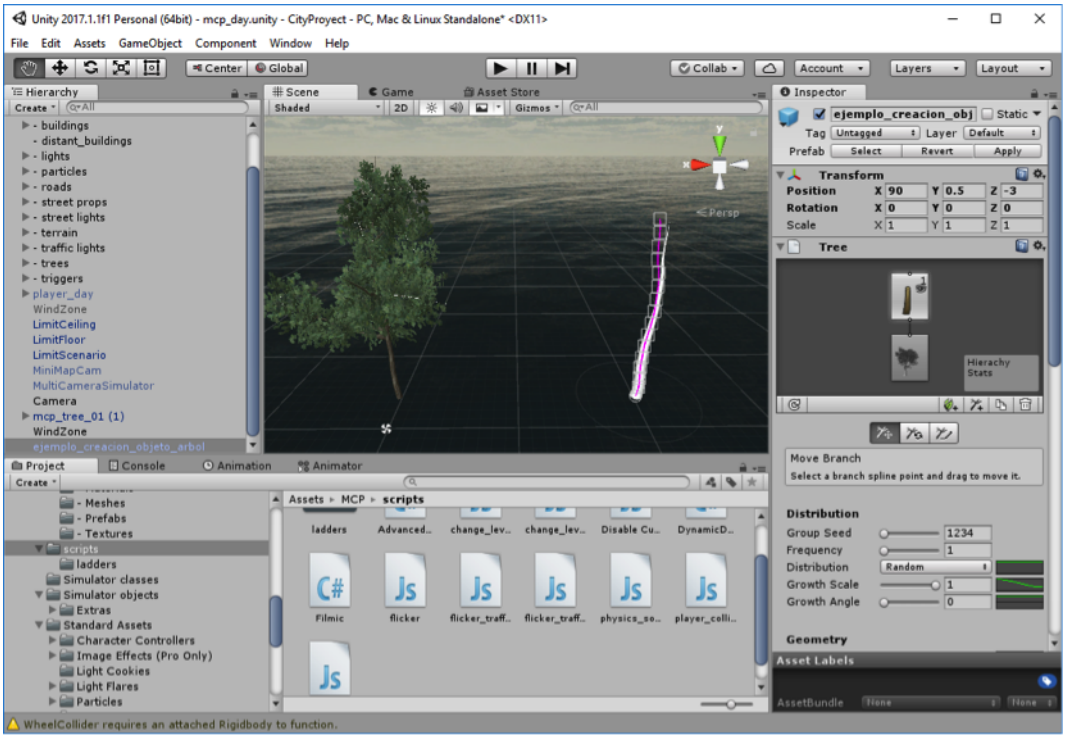


Figura D.40: Paso 2 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

En la propia jerarquía del árbol, podremos incluir tantas ramas y hojas a cada uno como mejor nos convenga, en este ejemplo hemos añadido ramas al tronco principal y hemos aumentado la frecuencia de aparición (Modificando el parámetro *Frecuency*).

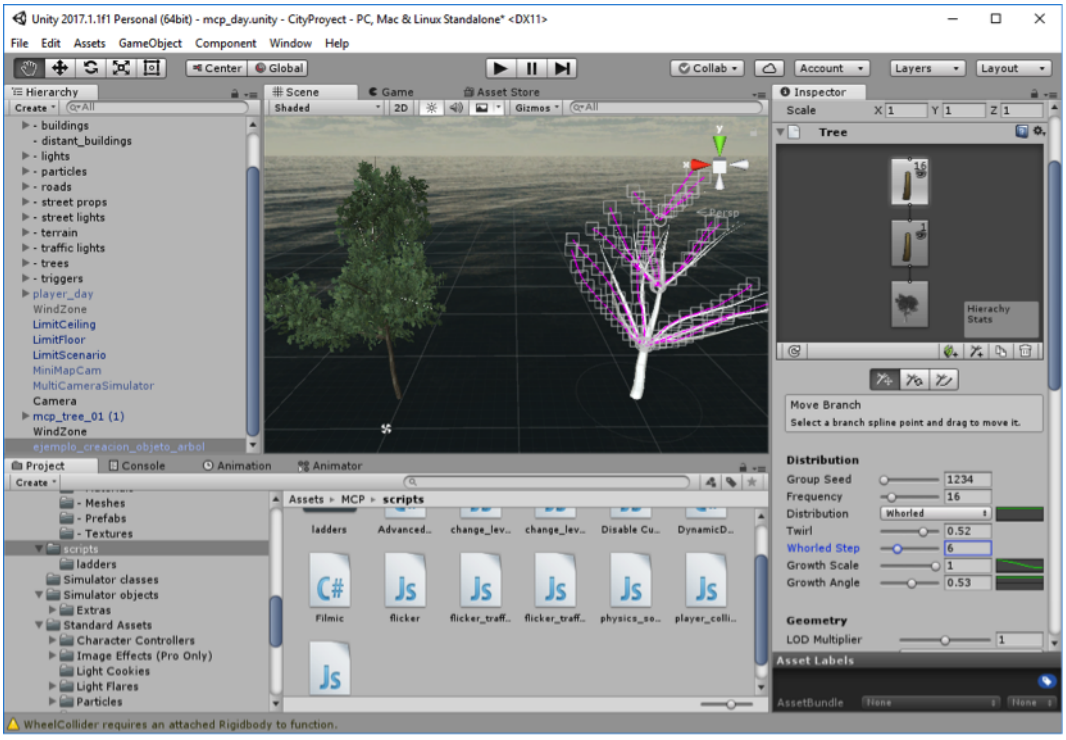


Figura D.41: Paso 3 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

A continuación hemos añadido hojas (dos tipos de hojas por ejemplo unas más grandes y menos frecuentes y otras mas pequeñas pero más frecuentes) a las ramas creadas anteriormente, tal y como se puede ver en la jerarquía de los elementos del objeto árbol.

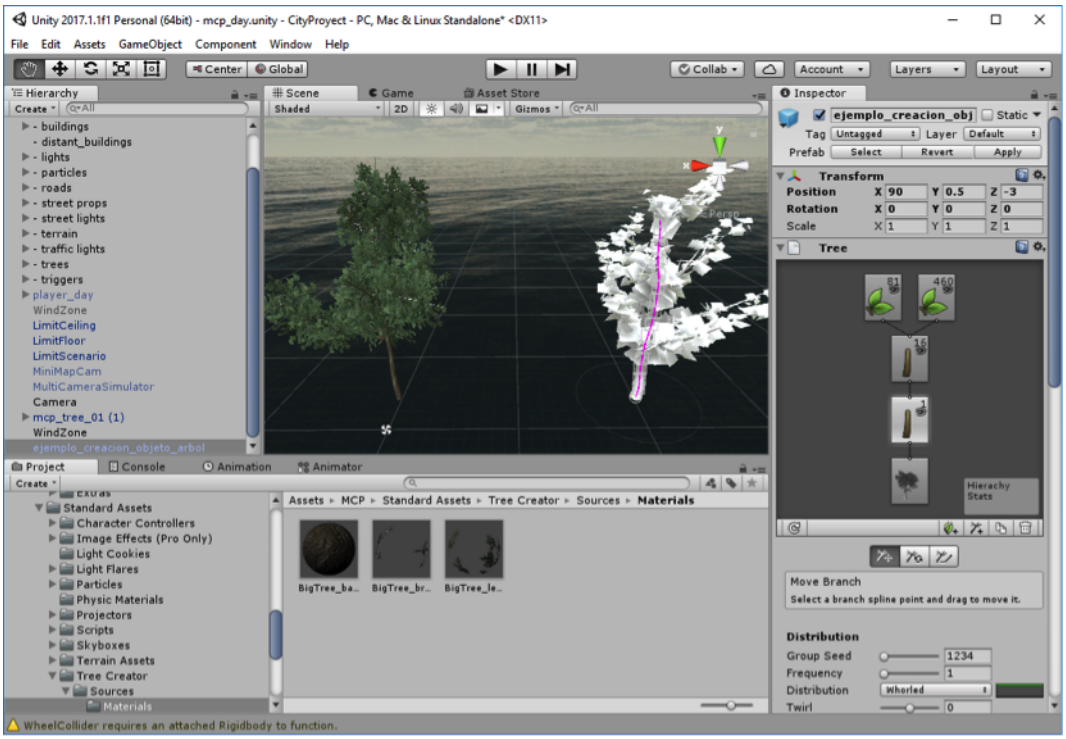


Figura D.42: Paso 4 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

A continuación, para este modelo geométrico primitivo de Unity (*Tree*) necesitaremos dos materiales, uno para el tronco y las ramas, simulando madera, y otro u otros para las hojas. Deberemos seleccionar cada elemento del árbol, y añadirle a cada uno un material. Para el tronco y las ramas les añadimos el material de madera (*BigTree bark*) y para las hojas el material (*BigTree leaves*).

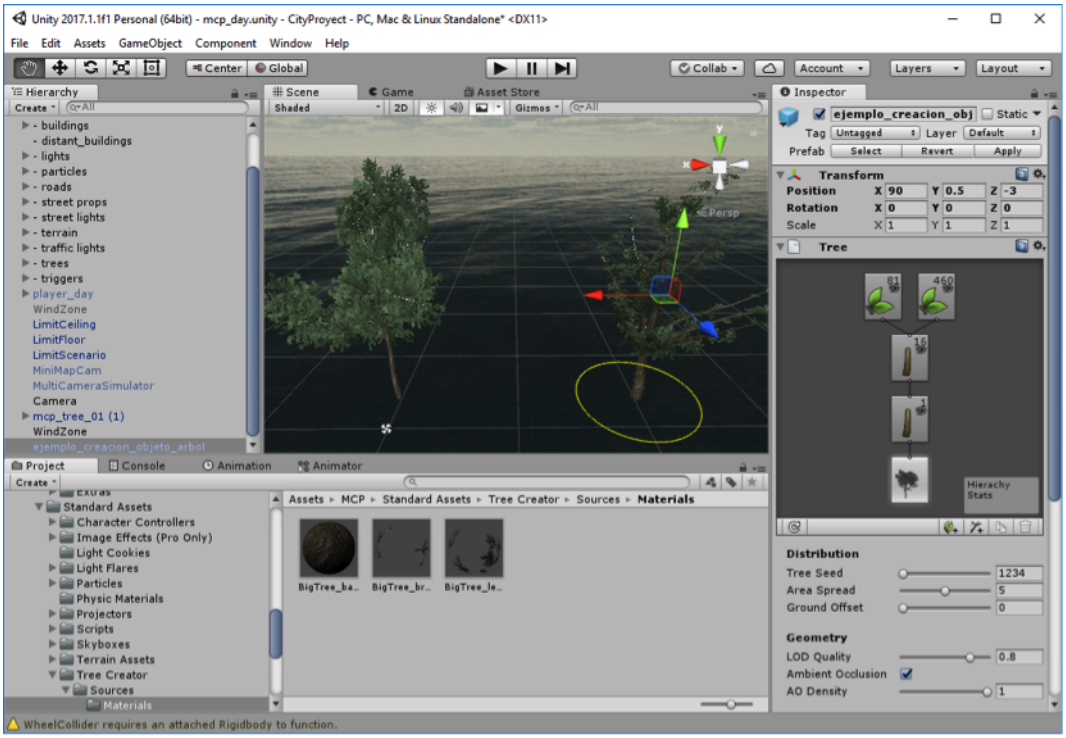


Figura D.43: Paso 5 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

Finalmente, se debe añadir un agente externo que simula el viento *GameObject > 3D Object > Wind Zone*. Este objeto especial interactuará con el controlador (que se encuentra en la propia herramienta o elemento de edición de árboles) de cada uno de los objetos árboles de la escena, otorgándoles más realismo, haciendo que no sean objeto estáticos, y provocando sombras dinámicas por el movimiento de sus ramas y hojas. Este objeto externo es el que transforma este objeto estático árbol, en un objeto periódico.

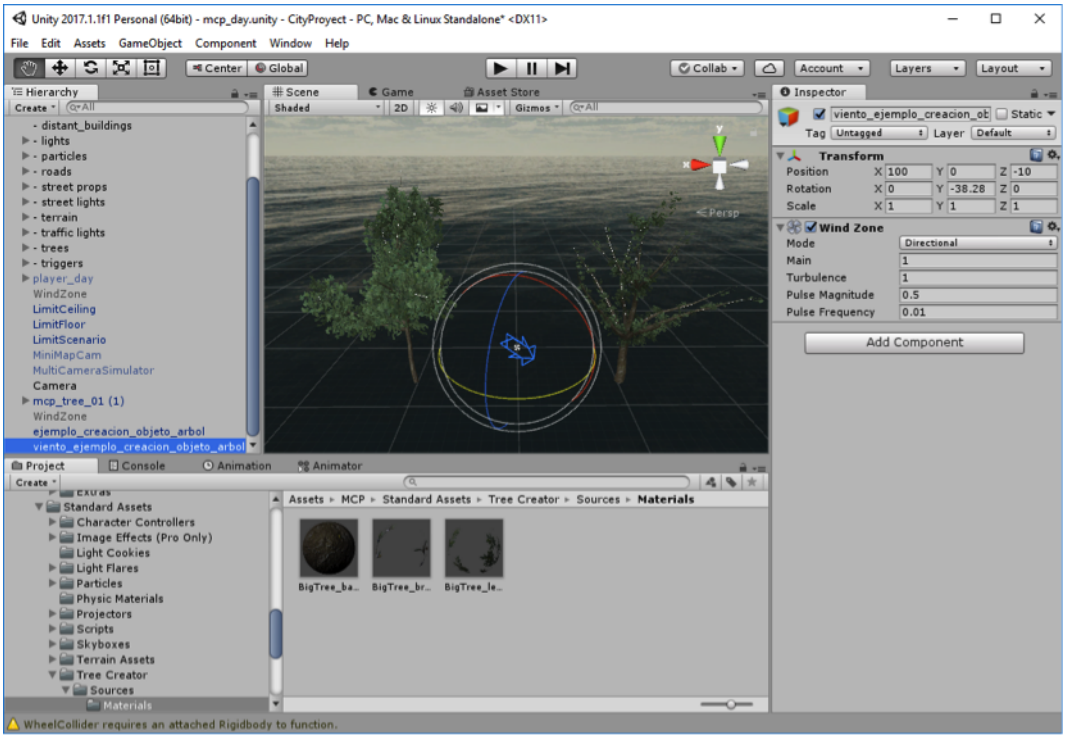


Figura D.44: Paso 6 del tutorial de creación de objeto periódico corpóreo (árbol con viento).

D.5.4. Objeto periódico incorpóreo

A continuación se muestra un ejemplo de cómo crear e instanciar un objeto periódico incorpóreo (**Fuente**) con un sistema de partículas (objeto primitivo de Unity *Particle System*). Los parámetros de edición del sistema de partículas podrán ser modificados a gusto del usuario:

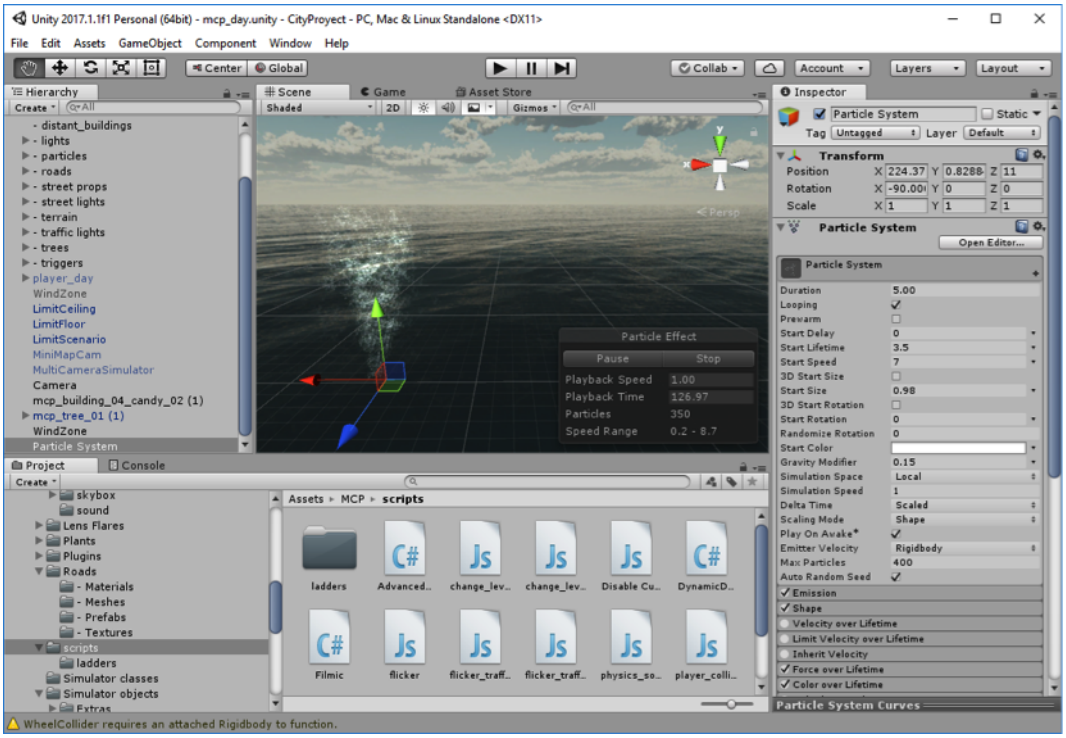


Figura D.45: Paso 1 del tutorial de creación de objeto periódico incorpóreo.

Para crear desde cero el objeto periódico **Fuente** debemos añadir un *GameObject* > *Particle System* y llamarlo como deseemos (por ejemplo: *ejemplo creacion objeto fuente*).

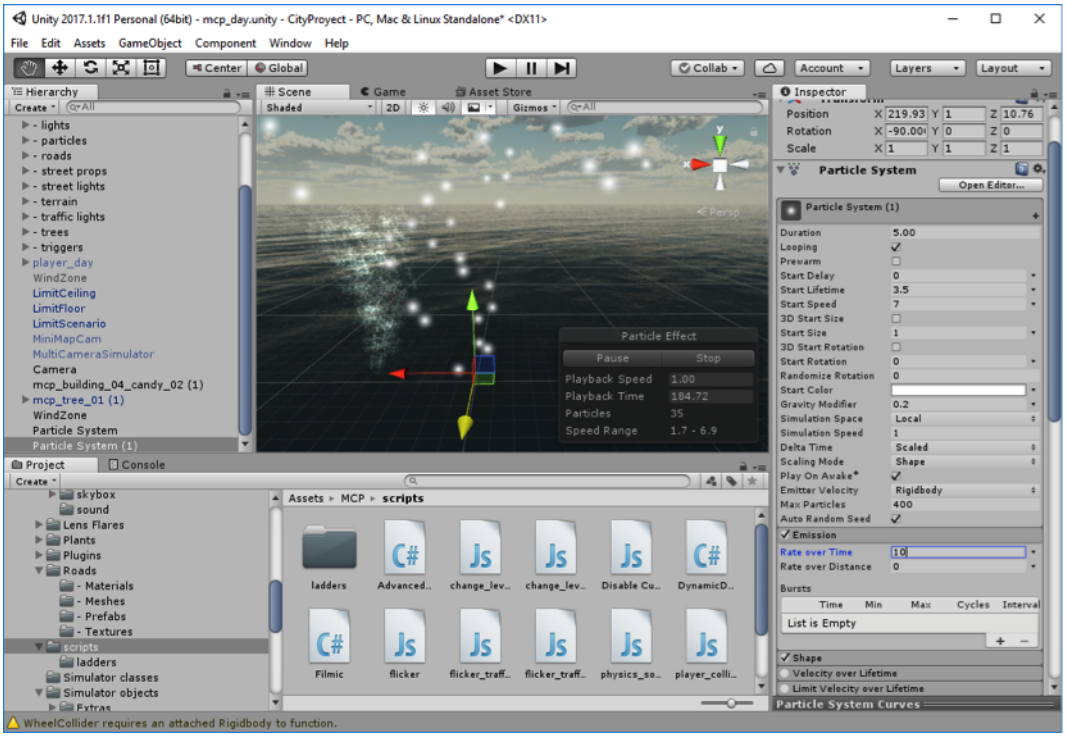


Figura D.46: Paso 2 del tutorial de creación de objeto periódico incorpóreo.

En el vídeo se podrá ver cada uno de los parámetros que fueron modificados secuencial-

mente, pero por no alargar el tutorial, se describen las configuraciones de algunos de los parámetros más representativos:

- Se ha modificado la emisión de partículas, cambiando el valor del parámetro *Rate over Time* en la pestaña *Emission* del objeto.

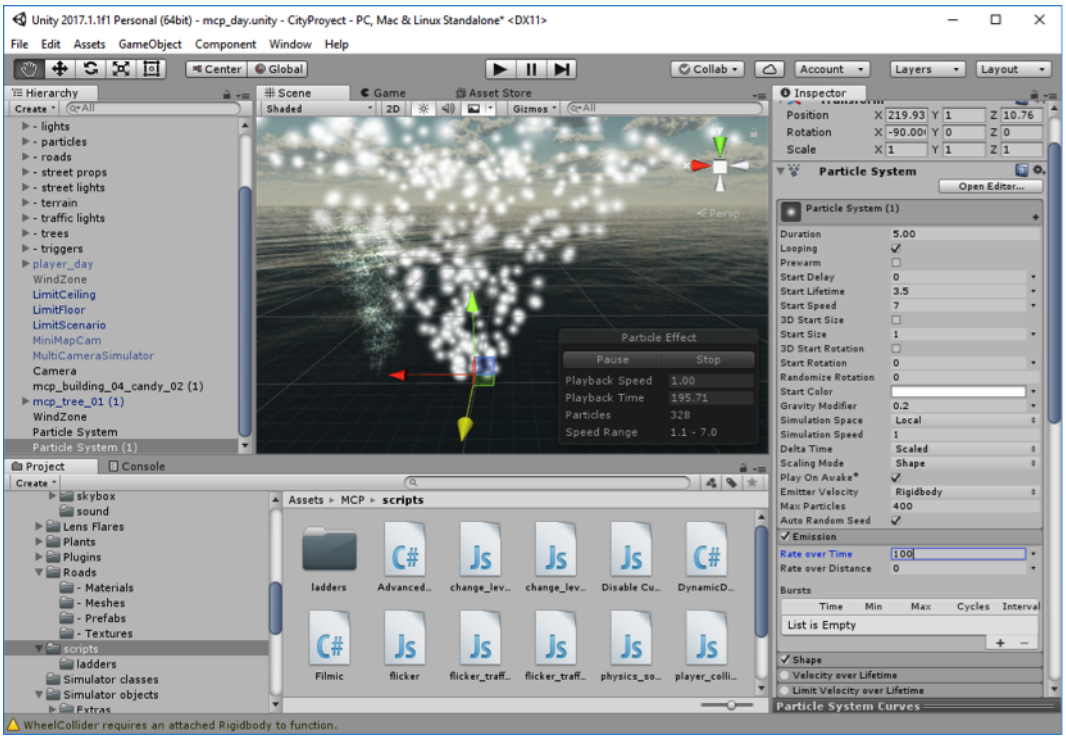


Figura D.47: Paso 3 del tutorial de creación de objeto periódico incorpóreo.

- Se ha modificado la forma del sistema de partículas, cambiando diferentes valores de parámetros como *Shape, Angle, Radius, Length* en la pestaña *Shape* del objeto.

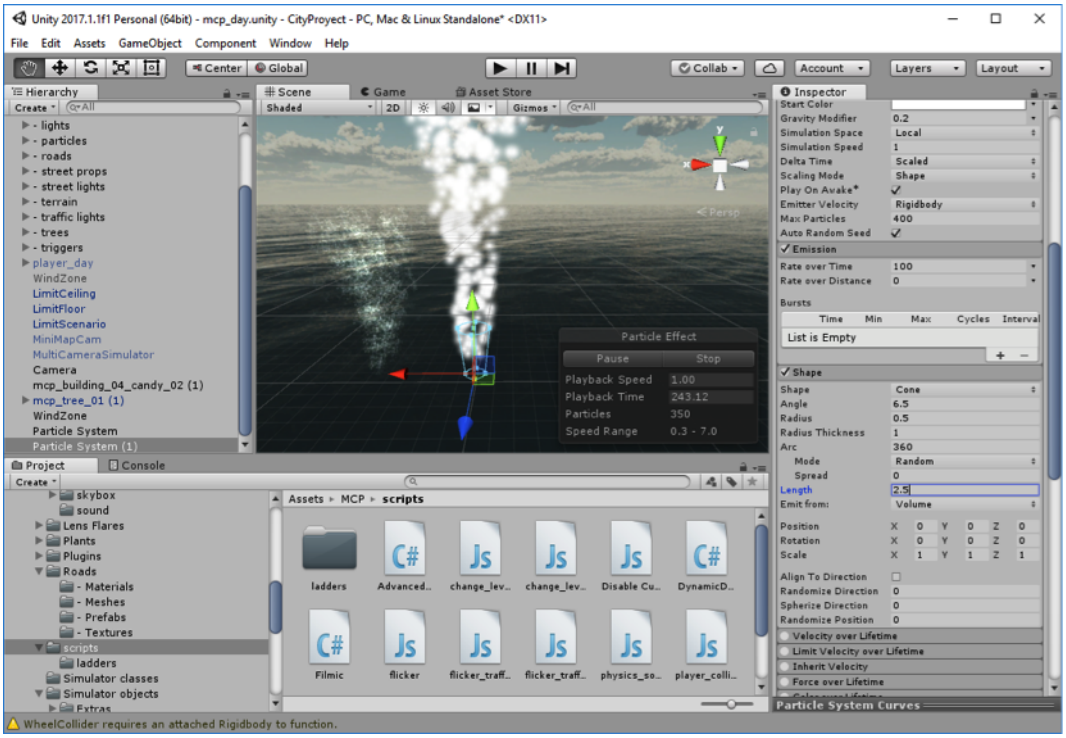


Figura D.48: Paso 3 del tutorial de creación de objeto periódico incorpóreo.

- Se ha modificado el comportamiento en cuanto a fuerza, color, tamaño y rotación de las partículas del a lo largo de su tiempo de vida, cambiando diferentes valores de parámetros en las pestañas *Force over Lifetime, Color over Lifetime, Size over Lifetime* y *Rotation over Lifetime* del objeto.

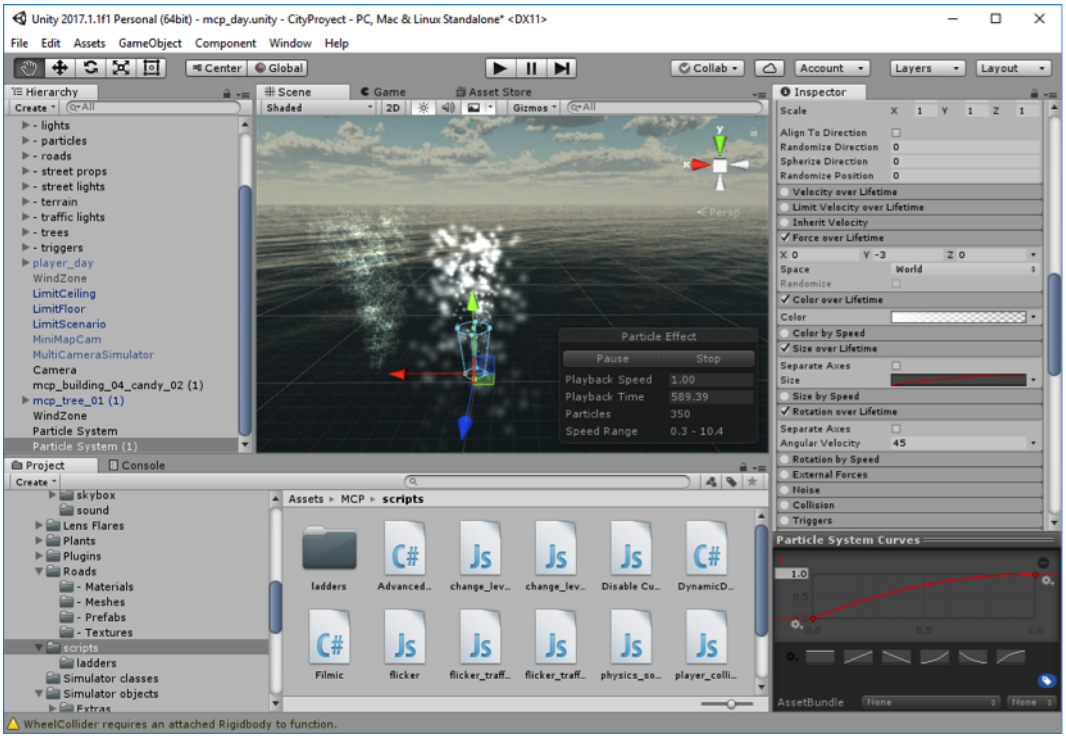


Figura D.49: Paso 4 del tutorial de creación de objeto periódico incorpóreo.

Finalmente, una vez hemos creado, definido, y configurado el sistema de partículas a nuestro gusto, deberíamos añadirle un material que simule el agua (*mcp fountain*).

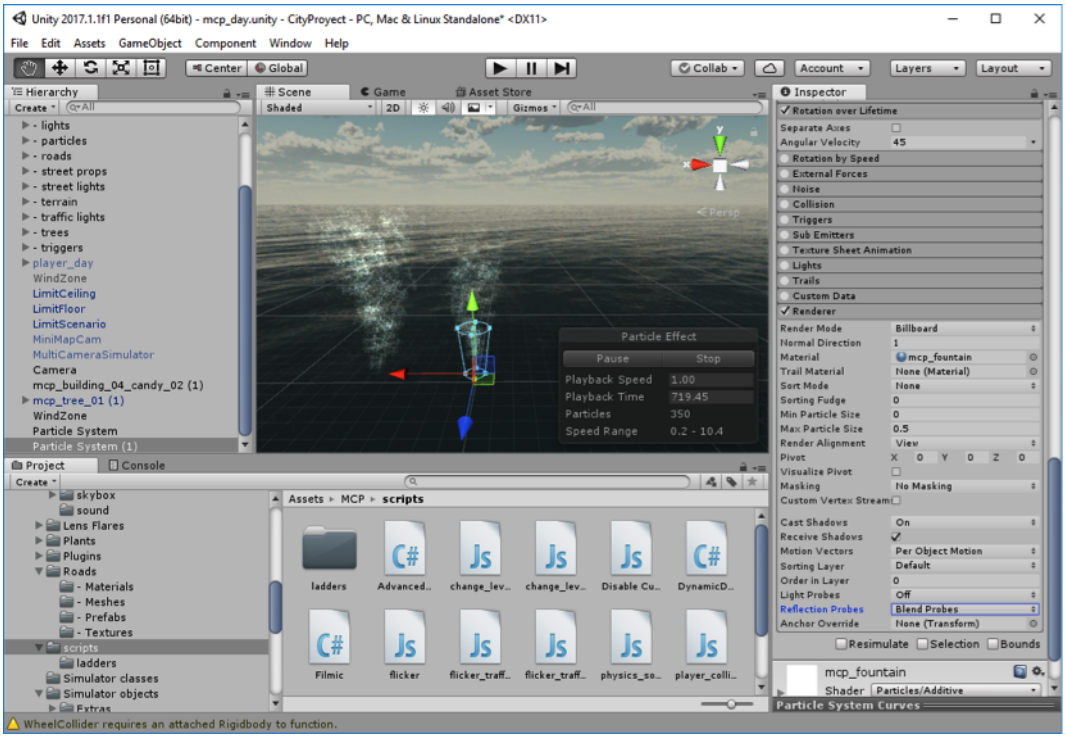


Figura D.50: Paso 5 del tutorial de creación de objeto periódico incorpóreo.